The dissertation of Jun Xu was reviewed and approved* by the following:

Peng Liu
Professor of Information Sciences and Technology
Dissertation Advisor, Chair of Committee

Sencun Zhu
Associate Professor of Computer Science and Engineering

Xinyu Xing
Assistant Professor of Information Sciences and Technology

Trent Jaeger
Professor of Computer Science and Engineering

Andrea Tapia
Associate Professor of Information Sciences and Technology
Director of Graduate Programs, College of Information Sciences and Technology

*Signatures are on file in the Graduate School.
Abstract

Software systems are expanding into every aspect of human society. Accompanying this expansion comes a substantial growth of motivated adversaries and sophisticated attacks. This pair of impulses make it imperative to secure software systems. To the security of software systems, a fundamental threat is vulnerability — a type of defect that allows adversaries to exploit for malicious intentions.

The battle against software vulnerabilities started two decades ago. Recently, the security community has been developing a consistent philosophy. It starts with vulnerability discoveries during product development and in-house testing. These are augmented by re-engineering the software systems to enforce run-time protections. These two lines of technique mitigate a great number of vulnerabilities, but they cannot resolve all of them. The reason behind is that vulnerability discovery does not scale well to large software and complicated vulnerabilities while in-depth run-time protection incurs performance overhead that goes beyond practical acceptance. This results in the practice that a substantial number of vulnerabilities are shipped to end users and we have no corresponding counteractions.

Among those unresolved vulnerabilities, there is an interesting observation — when those vulnerabilities are triggered either during exploit tests by attackers or normal operations by benign users, the software often runs into failure. The most common type of failure is software crash. According to Microsoft, it observes millions of crashes every day. Among the root causes of those crashes, nearly 10% are vulnerabilities. My dissertation research is inspired by this practice and explores to identify unresolved vulnerabilities with automated software crash diagnosis.

After a software has crashed, it typically leaves behind a snapshot of its crashing state in the form of a core dump. I design and implement CREDAL, an automatic diagnosis tool, to combine information in the core dump and source code of the crashed program to provide informative aid in tracking down the crash causes. CREDAL is featured with the capability to analyze crashes due to a common type of vulnerability known as memory corruption. For a core dump carrying corrupted memory,
systematically analyzes the core dump and identifies the crash point and stack frames. Further, CREDAL pinpoints the objects holding corrupted data using the source code along with the stack frames. To assist in tracking down the root cause, CREDAL also performs analysis and highlights the source code fragments responsible for the memory corruption.

The development of CREDAL carries two assumptions — source code is available and the crash occurred in a random exercise scenario. Because of that, CREDAL may experience usability and reliability problems. To address those shortcomings of CREDAL, I then designed POMP to locate the vulnerabilities behind software crashes, even when the source codes are unavailable and the crashed execution was under attack. POMP leverages a hardware feature on recent generations of Intel processors, Processor Tracing (PT), to trace the software execution and it includes the trace in the core dump. Along with the execution trace, POMP introduces a new reverse execution mechanism to construct the data flow prior to the crash. POMP then performs a backward taint analysis and highlights those instructions that actually pertain to the vulnerability, making the diagnosis more effective.
# Table of Contents

List of Figures ix  
List of Tables xi  
Acknowledgments xiii  

## Chapter 1  
**Introduction**  
1.1 Motivation ................................................. 1  
1.2 Techniques Overview ....................................... 3  
1.2.1 Locating a Memory Corruption Vulnerability with A Core Dump  
1.2.2 Postmortem Program Analysis with Hardware Enhanced PostCrash Artifacts ................................................. 6  
1.3 Summary of Contributions .................................... 9  

## Chapter 2  
**Related Work**  
2.1 Vulnerability Discovery ..................................... 10  
2.2 Run-time Mitigation ......................................... 14  
2.2.1 Run-time Detection ....................................... 14  
2.2.2 Run-time Protection ....................................... 15  
2.2.1.1 Address Sanitizing ...................................... 14  
2.2.1.2 Dynamic Tainting ...................................... 15  
2.2.2.1 Address Space Randomization.  
2.2.2.2 Control Flow Integrity  
2.3 Software Crash Diagnosis .................................... 16
Chapter 3
CREDAL: Towards Locating a Memory Corruption Vulnerability with Your Core Dump
3.1 Background and Introduction ........................................... 22
3.2 Problem Scope .............................................................. 25
   3.2.1 Threat Model .......................................................... 26
   3.2.2 Motivating Example .................................................. 26
3.3 Overview ................................................................. 29
   3.3.1 Design Principle ....................................................... 30
   3.3.2 Technical Approach .................................................. 30
3.4 Design ................................................................. 32
   3.4.1 Discovering Crashing Thread ...................................... 33
   3.4.2 Tracking down Crash Point ........................................ 33
   3.4.3 Identifying Stack Trace ............................................. 37
   3.4.4 Discovering Data Dependency Mismatch ......................... 38
3.5 Implementation ......................................................... 40
3.6 Case Study .............................................................. 42
   3.6.1 Setup ................................................................. 43
   3.6.2 Results ............................................................... 45
      3.6.2.1 Pinpointing Crash Point & Stack Trace ...................... 46
      3.6.2.2 Locating Vulnerability ....................................... 47
3.7 Discussion .............................................................. 49
3.8 Conclusion .............................................................. 50

Chapter 4
POMP: Postmortem Program Analysis with Hardware Enhanced Post-Crash Artifacts
4.1 Introduction and Background ........................................... 51
4.2 Problem Scope .............................................................. 55
   4.2.1 Threat Model .......................................................... 55
   4.2.2 Challenge ............................................................... 56
4.3 Overview ................................................................. 58
   4.3.1 Intel Processor Tracing ............................................. 58
   4.3.2 Objective ............................................................... 60
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3.3</td>
<td>Design Principle</td>
<td>60</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Technical Approach</td>
<td>61</td>
</tr>
<tr>
<td>4.4</td>
<td>Design</td>
<td>64</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Reverse Execution</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>4.4.1.1  Use-Define Chain Construction</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>4.4.1.2  Memory Alias Verification</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>4.4.1.3  Discussion</td>
<td>70</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Backward Taint Analysis</td>
<td>71</td>
</tr>
<tr>
<td>4.5</td>
<td>Implementation</td>
<td>73</td>
</tr>
<tr>
<td>4.6</td>
<td>Evaluation</td>
<td>74</td>
</tr>
<tr>
<td></td>
<td>4.6.1   Setup</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td>4.6.2   Experimental Design</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td>4.6.3   Experimental Results</td>
<td>77</td>
</tr>
<tr>
<td></td>
<td>4.6.4   Comparing POMP with CREDAL</td>
<td>80</td>
</tr>
<tr>
<td>4.7</td>
<td>Discussion</td>
<td>81</td>
</tr>
<tr>
<td>4.8</td>
<td>Conclusion</td>
<td>82</td>
</tr>
</tbody>
</table>

**Chapter 5**

**POMP-V: Improving Postmortem Program Analysis with Static Analysis**

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>84</td>
</tr>
<tr>
<td>5.2</td>
<td>Background of Value-set Analysis</td>
<td>88</td>
</tr>
<tr>
<td>5.3</td>
<td>Design</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td>5.3.1   Identifying Memory Regions</td>
<td>92</td>
</tr>
<tr>
<td></td>
<td>5.3.1.1  Stack</td>
<td>92</td>
</tr>
<tr>
<td></td>
<td>5.3.1.2  Heap</td>
<td>92</td>
</tr>
<tr>
<td></td>
<td>5.3.1.3  Global variables</td>
<td>93</td>
</tr>
<tr>
<td></td>
<td>5.3.1.4  Static variables and TLS variables</td>
<td>93</td>
</tr>
<tr>
<td></td>
<td>5.3.2   Value Set Analysis</td>
<td>94</td>
</tr>
<tr>
<td></td>
<td>5.3.3   Combining VSA and Reverse Execution</td>
<td>96</td>
</tr>
<tr>
<td></td>
<td>5.3.4   Further Improvements</td>
<td>97</td>
</tr>
<tr>
<td></td>
<td>5.3.4.1  Constancy of frame pointers</td>
<td>97</td>
</tr>
<tr>
<td></td>
<td>5.3.4.2  Non-writable memory.</td>
<td>98</td>
</tr>
<tr>
<td>5.4</td>
<td>Implementation</td>
<td>98</td>
</tr>
<tr>
<td>5.5</td>
<td>Evaluation</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>5.5.1   Evaluation Settings</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>5.5.2   Utility Evaluation</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>5.5.3   Efficiency Measurement</td>
<td>102</td>
</tr>
<tr>
<td>5.6</td>
<td>Discussion and Future Work</td>
<td>104</td>
</tr>
<tr>
<td></td>
<td>5.6.1   Memory Corruption</td>
<td>104</td>
</tr>
<tr>
<td></td>
<td>5.6.2   Multiple Threads</td>
<td>104</td>
</tr>
</tbody>
</table>
List of Figures

3.1 The status of the crashing stack and processor registers after exploiting the overflow vulnerability specified in CVE-2013-2028. For simplicity and demonstration, the stack protector has been disabled. .......................... 29

3.2 The snapshot of a crashing program after exploiting its overflow vulnerability. For simplicity and demonstration, the stack protector has been disabled and we convert the instruction on a binary into the statement in C programming language. .......................................................... 31

3.3 Stack pointer verification. For simplicity and demonstration, the return address points a line of C programming code converted from the instruction on a binary. .......................................................... 36

3.4 An inter-procedural control flow graph and data dependency constraints. Note that scanf() is a call to an external library. For simplicity and demonstration, we do not unfold this call. The list of constraints defines data dependency e.g., a@foo=1 indicates variable a in function foo should be equal to 1 at the crash; d@foo=[valid] indicates pointer d should be valid at the crash. .......................................................... 38

4.1 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. ............... 57

4.2 Example of trace generated by PT (with kernel tracing disabled). The left part shows the instruction sequence and the right part presents the corresponding PT trace. .......................................................... 59

4.3 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. .......................................................... 64
4.4 A use-define chain with one intervening tag conservatively placed. The tag blocks the propagation of some data flows. Note that the red “X” represents the block of a data flow. ................................................. 66

4.5 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 “X” represents the block of a data flow. ................................................................. 69

5.1 The toy assembly code. .......................................................... 87

5.2 Alias matrix identified by VSA. ‘0’, ‘1’ and ‘?’ represent non-alias, alias and may-alias relationships respectively. ......................... 88

5.3 A-locs and value-sets corresponding to complete and incomplete traces. 90

5.4 Examples of memory references to different regions ................. 91

5.5 Measurement of recovered memory addresses. HT refers to reverse execution with hypothesis testing. VSA indicates reverse execution with VSA. VSA+HT shows reverse execution with one round of VSA plus HT while Fix Point shows reverse execution that follows Algorithm 1 to reach a fixed point. Ground Truth refers to reverse execution with ground truth of all alias relations. ............................................ 102
List of Tables

3.1 A code fragment from Nginx-1.4.0 vulnerable to an integer overflow (CVE-2013-2028). .................................................. 27

3.2 The list of the program crashes corresponding to memory corruption vulnerabilities. CVE-ID and EDB-ID specify the IDs of the CVE and corresponding PoC, respectively. EIP indicates the validity of the program counter at the crash. # of Frame and # of Functions describe the number of stack frames CREDAL identifies as well as the number of functions one needs to examine when locating the corresponding vulnerability. The numbers in Area (LOC) indicate the lines of code corresponding to data dependency mismatch. ........................................ 45

4.1 A toy example with a stack overflow defect. .......................... 55

4.2 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. ........................................... 75

4.3 Diagnosis performance comparison between CREDAL and POMP. 80

5.1 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-V reversely executed. Root Cause VSA-POMP shows if we can capture the root cause when only using VSA for alias verification while Root Cause HT-POMP shows if we can capture the root cause when only using hypothesis testing for alias verification. 99
5.2 Measurement of reverse execution efficiency. HT refers to reverse execution with hypothesis testing. VSA indicates reverse execution with VSA. VSA+HT shows reverse execution with one round of VSA plus HT while Fix Point shows reverse execution that follows Algorithm 1 to reach a fixed point.
I would like to express my special appreciation and thanks to my advisor, Dr. Peng Liu. His advice on my research as well as on my career has been invaluable. In particular, I would like to thank him for encouraging my research and for allowing me to grow as an independent researcher. He works so hard to give us strong support and great freedom. His insights, encouragements and guidance are one of the most valuable things I obtained during this journey. I sincerely wish that in the future, I can do as he has always been doing.

I give special thanks to my co-adviser, Dr. Xinyu Xing. His generous support covers almost every aspect of doing research and being a professional scholar. His advising ranges from high level instructions such as research philosophy to guidance as detailed as paper writing. I truly appreciate his invaluable mentoring and advice for my research and career.

I would also like to express my sincere gratitude to my committee members, Dr. Sencun Zhu and Dr. Trent Jaeger for serving as my committee members even at hardship. Without the all-sided helps and suggestions from them, I would not have achieved this research.

I want to give many thanks to all of my research collaborators. Particularly, I want to thank Dr. Jimmy Su and Dr. Ping Chen at JD.com, Dr. Lenx Wei at Baidu.com, Dr. Ahmd Azib and Dr. Wenbo Shen at Samsung, Dr. Long Lu and Yaohui Chen at Northeastern University, Dr. Fengwei Zhang and Zhenyu Ning at Wayne State University, Dr. Bo Chen at Michigan Technological University, and Dr. Bing Mao, Dongliang Mu as well as Liang Deng from Nanjing University.

Finally, I would especially like to thank my family. My beloved wife, Mi, has been extremely supportive in the journey towards my doctoral degree and has made countless sacrifices to help me get to this point. My child, Linus, has continually motivated me to finish my degree with faith. My parents, Fuyun and Yihua, deserve special thanks for their continued support and encouragement. Without such a team, I would not be this place today.
This dissertation study was supported by ARO W911NF13-1-0421 (MURI), NSF CNS-1422594, NSF CNS-1505664, ARO W911NF-15-1-0576, ONR N00014-16-1-2265 and NIETP CAE Cybersecurity Grant. Any opinions, findings, and conclusions or recommendations expressed in this dissertation are those of the author and do not necessarily reflect the views of the NSF, ARO and ONR.
Chapter 1  
Introduction

1.1 Motivation

Around thirty years ago, our society started research on defense against cyber attacks. It has already provided us with a full stack of security solutions, including firewalls, intrusion detection systems and anti-virus engines. These solutions are protecting us from almost every aspect. However, our cyber space is not becoming safer. In May 2017, the WannaCry ransomware attack [1] compromised thousands of computers and caused four billion dollars of losses all over the world. In fact, there are many more attacks like WannaCry, such as the HeartBleed attack [2] and the Stuxnet attack [3].

Behind those cyber attacks is a common procedure. The attacker first finds a software vulnerability and makes an analysis. Then she develops an exploit and tests it on a small set of targets. After confirming stability of the exploit, she will launch the final attack and propagate the infections to a large scale. However, our society mitigates those attacks passively. Only after people observe the consequence of an attack, they develop solutions. This has become the root cause of today’s cyber insecurity.

Fortunately, security people have been re-thinking about this problem and have realized that the fundamental threat is software vulnerability — A type of defect in software systems that allows adversaries to exploit for malicious intentions. In recent years, the security community has been developing a proactive philosophy on combating
software vulnerabilities. It starts with vulnerability discovery during development and in-house testing. These are augmented by re-engineering the software systems to make them safer. The two lines of efforts have led to remarkable progress, while they are insufficient to fully resolve the problem.

To discover vulnerabilities, we usually do static analysis [4] or dynamic testing [5]. Static analysis inspects the code without execution to find vulnerabilities while dynamic testing runs the software and expects to trigger vulnerabilities. They have been widely used and resulted in identification of a great many of vulnerabilities. However, they are unable to cover all the vulnerabilities, because static analysis produces low precision in dealing with large software and dynamic testing can only reach a limited execution space.

To make software systems safer, we re-engineer them to enforce run-time detection [6, 7] or protection [8]. In practice, we only deploy preliminary versions of those techniques, as their in-depth versions introduce unacceptable costs. Although those preliminary versions can prevent exploits against less powerful vulnerabilities, they can easily be bypassed in cases where the vulnerabilities are expressive [9, 10].

In conclusion, we are still unable to resolve a great number of vulnerabilities. Among those vulnerabilities, there is an interesting observation. That is, when those vulnerabilities are triggered either during normal operations by benign users or exploit tests by attackers, the software often runs into failure. The most common type of software failure is software crash. According to Microsoft, they see millions of software crashes everyday and behind these crashes, nearly 10% are due to vulnerabilities [11]. Efficient and effective diagnosis of software crash will promptly identify the vulnerabilities behind and promote instant mitigation. This will significantly complement our battle against software vulnerabilities.

When software crashes happen today, we first triage them into groups based on their symptoms [12–14]. Then for each group, we mainly perform manual analysis to understand the root cause — In particular those vulnerabilities behind. Apparently, it is practically impossible to employ manual efforts on dealing with millions of crashes. My
dissertation research has been motivated by this practice and endeavoring to change it. I develop techniques to automate the diagnosis of software crashes that are caused by vulnerabilities.

1.2 Techniques Overview

Generally speaking, diagnosis of vulnerability-induced software crash is challenging. First, the majority of software crashes happen in-field, of which the corresponding inputs are unavailable. This makes conventional analysis techniques, such as interactive debugging, inapplicable. We necessarily need invention of new techniques. Second, the information left over by a software crash is very limited. It usually only includes a snapshot of the memory and registers. Leveraging such information to track down the root cause is substantially difficult. Third, vulnerabilities being triggered often leads to unintentional modification to the memory and thus compromises the data left behind the crash. This leaves a significant challenge for identifying useful information.

This dissertation research explores techniques to overcome the above challenges. At the high level, my techniques combine heterogeneous sources of information produced by a software crash and follow a general procedure to pinpoint the vulnerabilities. To be more specific, I start with reconstructing the control flow, leveraging program analysis with information left in the memory snapshot. Along with the control flow, I will attempt on re-building the data flow. Ultimately I will seek for security violations on the data flow, which are usually the locations of vulnerabilities. Considering the data I use may have been corrupted, I also design solutions to ensure the fidelity of those data and the trustworthiness of my analysis.

Guided by the above idea, I developed two techniques and designed corresponding platforms to implement those techniques. These techniques are featured with four properties. First, they are designed to perform diagnosis on software crashes that occur in filed. This enables them to cover the major families of software crash. Second, they
are particularly capable of locating memory corruption vulnerabilities from crashes of software developed with C/C++ language. This is an unresolved challenge in software crash diagnosis. Third, they support different types of analysis scenario. In particular, they can work with or without source code and they can also deal with software crashes due to either benign operations or exploit attempts. This significantly enlarges their generality. Fourth, their analysis modules are aware of the possibility of data corruption and can ensure the correctness of analysis. This brings the platforms high reliability. In the following, I will give a more detailed introduction of the two techniques and the corresponding platforms.

1.2.1 Locating a Memory Corruption Vulnerability with A Core Dump

After a software defect is triggered, and a software has terminated abnormally, it typically leaves behind a snapshot of its crashing state. In general, the snapshot of a crashing state is organized in the form of a core dump, which oftentimes contains the crashing program stack, the final values of local and global variables, and the final values of processor registers. Since a core dump carries certain clues as to a program crash, commercial software vendors oftentimes utilize it to facilitate failure diagnosis and classify crashes likely caused by the same defect [15–17]. While shown to be effective in triaging software crashes, existing technical approaches (e.g., [17, 18]) are less likely to be effective in identifying some program faults, particularly memory corruption vulnerabilities (e.g., buffer overflow and use after free).

A memory corruption vulnerability is a special type of faults in software that could lead to unintentional modification to the content at a memory location and thus compromise the data dependency of a running program. As such, a core dump may carry a certain amount of corrupted data when a memory corruption vulnerability is triggered and incurs a program crash. Since corrupted data can be anywhere in the memory, it leaves a significant challenge for identifying useful diagnosis information.
In my dissertation research, I develop CREDAL, an automatic diagnosis tool to assist software developers in tracking down software faults, particularly memory corruption vulnerabilities. To deal with the challenges that memory corruption introduces to core dump analysis, technically, CREDAL leverages the source code of the crashing program to enhance core dump analysis.

Since the state of the program at the crash is an almost-necessary starting point [19] and stack traces can potentially narrow down the list of candidate files that are likely to contain software defects [20], CREDAL first identifies the crash point and attempts to restore the stack trace of a crashing program.

In general, it is easy to identify a crash point. When a program has crashed and terminated unexpectedly, the final value of the program counter typically indicates the crash point. However, memory corruption may manipulate the program counter, making it point to an invalid instruction. (e.g., overwriting a return address on the stack with a non-executable memory location). To address this problem, CREDAL checks the validity of the program counter. For the invalid program counter, CREDAL restores its value by analyzing the remnants on the stack. More specifically, CREDAL first attempts to identify the function which was just called but silently returned before the crash, in that this function carries the information about its parent (i.e., the crash function). Using this function, CREDAL then locates the crash function as well as the crash point within it. In recovering the crashing stack, CREDAL makes conservative inference using the restored program pointer along with call frame information [21].

As is discussed above, memory corruption can manipulate the content at a certain memory location, which may result in the violation of data dependency. Intuition suggests that highlighting data dependency mismatch seems informative for failure diagnosis. As a result, we further augment CREDAL with the ability of specifying data dependency mismatches at the source code level. In particular, CREDAL identifies the variables – the values of which in memory mismatch the data dependency of the crashing program – and highlights the source code corresponding to the mismatches. Technically speaking,
CREDAL first constructs an inter-procedural control flow graph based on the stack traces restored as well as the source code of the crashing program. Then, it performs inter-procedural points-to analysis and reaching definition analysis to discover the mismatch in variable values and pinpoint the code fragments corresponding to the mismatch (which are the locations of vulnerabilities).

1.2.2 Postmortem Program Analysis with Hardware Enhanced PostCrash Artifacts

The development of CREDAL carries two assumptions - (1) source code is available and (2) the crash occurred in a random exercise scenario. Because of that, CREDAL may experience usability and reliability problems. To be specific, under the circumstances that source code is not provided, CREDAL cannot be used; When the software crashed because of attacks, CREDAL may produce incorrect analysis. To address those shortcomings of CREDAL, I then designed POMP [22] to locate the root causes of software crashes, even when the source codes are unavailable and the crashed execution was under attack.

Recently, the advance in hardware-assisted processor tracing significantly ameliorates this situation. With the emergence of Intel PT [23] – 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. A post-crash artifact typically contains a large amount of instructions. Even though it carries execution

\footnote{By a post-crash artifact, without further specification, we mean a core dump including both the...}
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, I 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 data integrity in crashing memory [18, 24] or heavily relies upon the capability of recording critical objects in memory [25–28]. 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.

Hypothesis testing resolves memory alias without errors and POMP demonstrates its utility in handling thousands of instructions. However, it may encounter reduced utility when dealing with long instruction traces. The reason behind is that the information useful for verifying the hypotheses reduces as the trace increases. Gradually, hypothesis testing snapshot of crashing memory and the instructions executed prior to the crash.
becomes insufficient to resolve alias relations. Taking the study shown in Table 4.2 for an example — The number of unknown memory addresses enlarges as the trace grows. Such a limitation may prevent POMP from producing informative results in more complicated cases. In addition to that, POMP is designed to perform recursive hypothesis testing. This incurs exponentially increasing cost. For instance, diagnosis in the case of unrar takes around 6 hours, which is largely beyond the industrial requirement that a case should be processed in a time slot up to seconds.

Over the past decades, there have been many works proposed to perform alias analysis at the binary code level (e.g., [29–31]). Among all these efforts, value set analysis (VSA) is the most effective and efficient technique [32, 33]. At a high level, VSA subdivides the memory regions into variable-like entities, based on how memory is accessed, and derives an over-approximation of the set of addresses on which variables span. With that, VSA allows two memory references at a program state to be interpreted with respect to the set of possible values that can arise at that state, and thus determines whether they are an alias pair pointing to the same memory cell.

In this dissertation, I enhance POMP with the VSA analysis. Technically speaking, I re-use the design of POMP to reconstruct the instruction trace. Against this trace, we perform VSA to obtain the value set pertaining to each memory access. Then we proceed to the reverse execution borrowed from POMP, but we take a different strategy to resolve memory alias — When seeing an uncertain alias, we first query the VSA results for answers. If the uncertainty persists, we switch to hypothesis testing.

The rationale behind the above hybrid design is two-fold. First, VSA purely relies on static information in the binary and such information rarely reduces as the trace becomes longer. As a result, it provides almost constant utilities, regardless of the trace length. This helps resolve a substantial amount of alias relations that hypothesis testing is unable to handle. In turn, resolution of those aliases compensates the information loss during the reverse execution, which can enable further progress. As we will demonstrate in Chapter 5, this hybrid design significantly improves the reconstruction of data flow.
Second, in addition to increasing the utilities, VSA also improves the efficiency of our diagnosis. Alias query based on the VSA results incurs cost that is negligible in comparison with hypothesis testing. By replacing a large amount of hypothesis testing with VSA queries, we can substantially reduce the diagnosis time.

1.3 Summary of Contributions

This dissertation research makes the following contributions.

- This dissertation research develops techniques to automate software crash diagnosis, which are the first techniques that can pinpoint vulnerabilities behind software crashes. The first technique, CREDAL, leverages the source code of the crashing program to enhance core dump analysis and provides useful information for software failure diagnosis; While the second technique, POMP, analyzes post-crash artifacts by reversely executing instructions residing in the artifact. The two techniques (1) support diagnosis on software crashes that occur in filed (2) can work with or without source code and they can also deal with software crashes due to either benign operations or exploit attempts (3) and are aware of the possibility of data corruption and can ensure the correctness of analysis.

- This dissertation research implements both CREDAL and POMP on Linux platforms. POMP has been open sourced at https://github.com/junxzm1990/pomp and it has been followed by multiple research groups to support related research.

- This research has demonstrated the effectiveness of CREDAL and POMP with well-sized case studies. It measured the utility of CREDAL in facilitating memory corruption vulnerability diagnosis by using 80 crashes attributable to 73 memory corruption vulnerabilities. Also, it evaluated the performance of POMP in facilitating software debugging by using various post-crash artifacts attributable to 31 distinct real world security vulnerabilities.
Chapter 2  
Related Work

In general, this dissertation research mainly seeks to battling cyber attacks via combating software vulnerabilities. Technically speaking, it endeavors to automating software crash diagnosis and locating the vulnerabilities behind. Therefore, the lines of research most closely related to this dissertation include vulnerability discovery, run-time vulnerability mitigation and software crash diagnosis. In this section, I summarize and discuss these works. To be more specific, for the works on vulnerability discovery and run-time mitigation, I mainly give an overview of them and explain their strengthens and potential future directions, since my research is mostly complementary to them. With regard to the techniques on software crash diagnosis, I will not only explain their details but also compare them with my dissertation research.

2.1 Vulnerability Discovery

It has been a long practice to develop automated techniques for vulnerability discovery. There have developed two major categories of such techniques, including *static analysis* and *dynamic testing*.
2.1.1 Static Analysis

Static analysis is the analysis of program without execution [4]. It reasons about the program behaviors for intentions such as error detection. Static analysis has high coverage of all possible execution scenarios. This, however, makes static analysis produce over-approximation. Hence, achieving a good trade-off between coverage and precision has long been a major challenge of static analysis. The recently developed and most popular frameworks that may facilitate vulnerability discovery include LLVM [34], SVF [35], and Dr.Checker [36].

LLVM [34] was initialized as a project to build a new compilation tool-chain at the University of Illinois. LLVM provides an architecture-independent, highly expressive and user-friendly Intermediate Representation (IR). Besides, it develops a comprehensive set of application interfaces to access, analyze and instrument the IR. This has made LLVM the most popular infrastructure for program analysis research. Recently, a group of widely used static analysis utilities have been integrated into LLVM, such as point-to analysis utilities [37].

SVF [35] is a framework built on the top of LLVM. It supports scalable and efficient inter-procedural static value-flow analysis. Technically speaking, SVF includes mainstream pointer-analysis algorithms into the framework, which provide all types of sensitivity. Using the point-to results, it constructs an inter-procedural SSA of the memory. In addition to that, SVF provides interfaces to obtain the def-use chains of both top-level and address-taken variables and format them as value flows. A user can then build different kinds of clients, which query the value flows and perform the required analysis (e.g., memory leakage detection).

Dr.Checker [36] is also a general static analysis framework. It provides interfaces for flow-sensitive, context-sensitive, and field-sensitive solutions for alias analysis and taint analysis. Differing from SVF, Dr.Checker always produces sound results and re-designs many standard algorithms to support the special requirements in dealing with kernel code.
2.1.2 Dynamic Testing

Dynamic testing executes the software with concrete inputs and examines the behaviors of the software. On observing exceptions, the testing tool reports an alert about existence of implementation flaws. Different from static analysis, dynamic testing has high precision in the sense that it rarely makes mistakes. However, dynamic testing can only cover a limited execution space, leading to low defect coverage. Therefore, a broadly acknowledged challenge of dynamic testing is to generate inputs which can efficiently achieve high coverage. Towards this goal, the community has been exploring different automated techniques, mainly from the perspective of fuzz testing and symbolic execution.

2.1.2.1 Fuzz Testing

Fuzz-testing (or fuzzing) is an automated software testing technique for unveiling various kinds of bugs in software. It provides invalid or randomized inputs to programs with the goal of discovering un-handled exceptions and crashes. This easy-to-use technique has now become the de facto standard in the software industry for robustness testing and security vulnerability discovery.

After the development for many years, gray-box fuzzing has been becoming the most popular one and has already demonstrated great effectiveness [38, 39]. A gray-box fuzzing tool start with feeding a seed input to the target software. Meanwhile, the fuzzing tool traces the software and records the code segments covered. Using the code coverage as feedback, the fuzzing tool mutates the seed input and produces new ones for further testing. This process repeats until code coverage converges to a fix point. Among all the grey-box fuzzing fuzzing tools, American Fuzzy Lop (AFL) requires essentially no a-priori knowledge to use and can handle complex, real-world software [39]. Therefore, AFL and its extensions have been widely adopted in practice, constantly discovering unknown vulnerabilities in popular software packages (such as...
nginx, OpenSSL and PHP). Many works followed AFL to make improvement from the perspective of effectiveness and efficiency.

For better fuzzing effectiveness, following works focus on improving seed generation and input scheduling. Skyfire [40] establishes a probabilistic context-sensitive grammar model by learning through a large corpus of valid inputs. It then uses the grammar to generate inputs that are accepted by target programs. Similarly, Godefroid et al. aid fuzzing with a grammar-based input generator [41, 42]. When there are plenty of inputs to select, the strategy to select seeds for the following runs is very critical for the efficiency of fuzzing test. AFL develops a scheduling algorithm in a round-robin flavor which prefers seeds that bring new edge coverage and take less time to run. Böhme et al. [43] propose to change that algorithm to prioritize inputs that follow less frequently visited paths. This strategy significantly accelerates the code coverage and bug discovery. Further, Vuzzer [44] shows that having the fuzzer focused on longer paths can improve the fuzzing yields.

2.1.2.2 Dynamic Symbolic Execution

Symbolic execution was initially developed for program verification [45]. It specifies symbolic inputs and interprets the program that receives the symbolic inputs. When conditional statements are reached, it forks the interpreter to follow both branches and adds the corresponding constraints to each branch. Theoretically, symbolic execution can explore all the possible execution paths. In practice, due to the tremendous amount of execution paths, symbolic execution may run into the problem of path-explosion and takes extremely long time to finish. Recently, symbolic execution has been applied and customized to explore vulnerabilities in software systems.

A series of systems have been developed for vulnerability discovery with or without source code [46–48]. Among them, KLEE [47] is the most famous symbolic executor for source-available programs while angr [48] is widely recognized for its ability to find vulnerabilities in binary code. Both of these two systems are well engineered
and can seamlessly support real world programs. However, neither of them can well address the path explosion problem and thus they are not really scalable enough to handle complicated software systems. Different from these two systems, CUTE [49] proposes to combine symbolic execution and concrete execution. The goal of CUTE is comprehensively explore a limited code segment which is led to by the concrete execution. CUTE can slightly mitigate the path explosion problem by compromising coverage.

2.2 Run-time Mitigation

Despite best efforts during software development and testing, vulnerabilities are inevitably missed in software and shipped to end users. To mitigate the risks of these vulnerabilities, plenty of research has been developing techniques to enforce run-time mitigation. Generally speaking, these techniques can be classified into two major groups. The first group adds into the software with extra modules to detect the triggering of vulnerabilities at run-time while the other group enforces the software systems to prevent vulnerabilities from being exploited.

2.2.1 Run-time Detection

2.2.1.1 Address Sanitizing

Address sanitizing is a technique to monitor and sanitize the memory accesses at run-time. It usually instruments the program code to detect memory access errors in type-unsafe programs.

AddressSanitizer is a widely adopted address sanitizing system, which can detect out-of-bounds accesses to various types of data, including heap, stack, and global data. It also isolates deallocated memory into a red-zone which helps detect use-after-free bugs. AddressSanitizer achieves comprehensiveness and higher efficiency than other similar techniques such as valgrind [50]. Up to present, AddressSanitizer
has been integrated into mainstream compilers such as GNU GCC and LLVM and has identified hundreds of previously unknown bugs in large software systems.

### 2.2.1.2 Dynamic Tainting

Dynamic tainting is another major category of technique to detect memory errors when the software is running. This technique works by tracing the software execution and watching for violations of memory safety (such as propagating user-controlled inputs to function pointers). Among those techniques, TaintCheck is the earliest exploration and has been extensively followed. TaintCheck performs dynamic binary rewriting to tracking memory propagation and captures most types of vulnerability when they are exploited. However, due to its dynamic rewriting, TaintCheck introduces so much performance overhead that it is less practical for real world deployment. Motivated by this, the research works following TaintCheck, such as libdft [51] and TaintPipe [52], mainly focus on improving the analysis efficiency.

### 2.2.2 Run-time Protection

#### 2.2.2.1 Address Space Randomization.

When exploiting a memory corruption vulnerability, attackers have to know the addresses of critical code. Therefore, an appealing defense is to randomize the locations of program code such that the address identified by attackers will be invalid. A large number of address space randomization (ASR) techniques have been proposed, and they differ at the granularity such as segment offset level [53, 54], page level [55], function level [56], basic block level [57], and instruction level [58, 59]. Also, the randomization can occur at compile time (e.g., [60]) or load-time (e.g., [54, 55, 58]). However, most of the existing ASR only performs a one-time randomization. Their effectiveness has been challenged by sophisticated information-leakage assisted attacks such as the JIT-ROP attack [10, 61, 62].
To enhance the resilience to information leakage, several works [63–66] propose the re-randomization idea. Curtsinger et al. [64] propose a function level re-randomization technique which maintains high efficiency. Another work [63] targets at kernel protection and does fine-grained re-randomization on multiple objects (e.g., functions and stack). With the same goal, TASR [65] and Isomeron [66] defeat JIT-ROP by dynamically re-randomizing the code address space and execution path, respectively.

### 2.2.2.2 Control Flow Integrity

Control Flow Integrity (CFI) is a defense mechanism that prevents code reuse attacks from arbitrarily controlling program behavior [67–72]. In particular, it confines all control flow transfers to follow a valid path defined in a static control flow graph. Theoretically, CFI can thwart many types of control-flow hijacking attacks, ranging from trivial ROP/JOP to sophisticated ROP (like JIT-ROP and BROP [61]). CFI has been proposed for many years and it has been deployed in the real world [73]. However, as is demonstrated in [74, 75], fine-grained CFI provides strong security guarantee but at the same time introduces performance overhead significantly high. While recent research indicates coarse-grained CFI can significantly improve performance, it also shows CFI is vulnerable to sophisticated attacks [70, 76–78].

### 2.3 Software Crash Diagnosis

Since more than 10 years ago, research on software crash diagnosis has started. It has accumulated plenty of solutions as well as systems. There are mainly five lines of research in this direction. In the following, I respectively summarize them and compare them with CREDAL and POMP.
2.3.1 Record and Replay

Software *record and replay* [79–82] is a powerful technique to debug crash during software development or testing. On running of a software, a record and replay tool can record all the execution states, including values of registers, status of memory, and external inputs from files or network. On occurrence of a crash, the software can be replayed to the error state based on the recordings, and investigated for the cause of crash. However, it is unrealistic to use this technique for analyzing software crashes from end users, as the recording scheme causes too much performance overhead. Different from record and replay, CREDAL and POMP only rely on information left behind by a software crash, introducing no intrusiveness in any type of software.

2.3.2 Program Instrumentation

To spot program faults, a large amount of research focuses on failure reproduction using execution traces (*e.g.*, [83–89]). Technically speaking, the typical approach along this line is to instrument a program, so that it can automatically generate execution traces at run time. By analyzing these execution traces, one can derive control and data flow and thus identify the faults in software. Since this run-time recording scheme provides more information about program execution, it is effective in locating program faults. In practice, this approach however has seldom been adopted presumably because many of these approaches introduce unacceptably high overhead during normal operation.

Many other works instead instrument programs to spot memory corruptions at run-time, such as AddressSanitizer [90], SoftBound [91], and Code-Pointer Integrity [92]. AddressSanitizer uses shadow memory to record whether a memory access is safe, and relies on instrumentation to verify the shadow memory in *load* or *store*; SoftBound inserts run-time bounds checks to enforce spatial safety using customized disjoint memory meta-data; Code-Pointer Integrity would detect when a code pointer is overwritten and terminate the execution. With Code-Pointer Integrity, code dump will be generated
before any illegal control flow transfer and thus, involves less uncertainty for analysis by CREDAL. These techniques aim at detecting corruptions instead of pinpointing the vulnerability areas. In practice, many of these approaches introduce high overhead during normal operation, which greatly affects their deployment.

Considering practicality, CREDAL and POMP do not instrument programs, nor rely upon the availability of existing program logging or execution traces. Rather, CREDAL and POMP facilitate program failure diagnosis by using more generic information (i.e., the core dump that operating system automatically captures every time a process has crashed or otherwise terminated abnormally).

### 2.3.3 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., [93–100]). 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. [98] proposed to use static backward analysis to reconstruct execution traces from a crash point and thus spot software defects, particularly typestate errors [101]. Similarly, Strom and Yellin [99] defined a partially path-sensitive backward data flow 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. also proposed a backward analysis technique for crash analysis [102]. In particular, they introduce an efficient algorithm that takes as input a crash point as well as a global 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 or event traces. While our work also uses stack traces for crash analysis, our approach is fundamentally different.
As is mentioned earlier, memory information might be corrupted when attackers exploit a program. Thus, the path analysis based on stack traces described in [102] fails because its effectiveness highly relies upon the stack integrity. In contrast with [102], CREDAL leverages the source code of a crashing program to enhance core dump analysis and pinpoints the code statements where a software defect is likely to reside.

2.3.4 Core Dump Forensics

Considering the low cost of capturing core dumps, prior studies [17, 18, 103–105] proposed to use core dumps to analyze the root cause of software failures. To facilitate software failure debugging, Polishchuk et al. [104] for example proposed a mechanism to reconstruct variable types from heap memory, and Salkeld and Kiczales [105] introduced a method to resurrect Java objects from a shadow heap dump. As part of POMP, I also restore memory information, but go beyond objects in memory at the time of the program crash.

Stepping over memory semantic reconstruction, Microsoft developed Windows Error Reporting (WER) service [17], which uses a tool — !analyze — to examine a core dump and determine which thread context and stack frame most likely caused the error. Although sharing the same goal as our work, !analyze cannot handle program crashes caused by security defects for the reason that the attacks may introduce memory corruption and processor register failures and the effectiveness of !analyze highly relies on these information.

In recent research, Wu et al. [103] proposed CrashLocator, a method to locate software defects by analyzing stack information in a core dump. In addition, Cui et al. [18] introduced RETracer, a system that reconstructs program semantics from core dumps and examines how program faults contribute to program crashes. More specifically, RETracer leverages a core dump along with a backward analysis mechanism to recover program execution status and thus spot a software defect. Since the effectiveness of both techniques highly relies upon the integrity of a core dump, and
exploiting vulnerabilities like buffer overflow and dangling pointers corrupts memory information, CrashLocator and RETracer fail to perform crash analysis.

Different from aforementioned core dump forensics, my approach can deal with both corrupted and un-corrupted core dumps and facilitate program failure diagnosis. To the best of my knowledge, CREDAL and POMP are the first works that analyze the core dump of a crashed program without the assumption of memory integrity.

2.3.5 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 [25–27] 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 [26, 27] 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 [106] to instrument C++ program in a way that it can store program states for reverse execution. Similarly, Sauciuc and Necula [107] 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 [18] and [24], 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 both techniques highly relies upon the integrity of a core dump, and exploiting vulnerabilities like buffer overflow and dangling pointers corrupts memory information, both of them fail to analyze software crash caused by malicious memory corruption.

Different from the research discussed above, the reverse execution technique introduced in POMP 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. Third, it is effective even though the memory carries corrupted data.
Chapter 3  
CREDAL: Towards Locating a Memory Corruption Vulnerability with Your Core Dump

3.1 Background and Introduction

Despite the best efforts of software developers, software inevitably contains defects. After a software defect is triggered, and a program has terminated abnormally, it typically leaves behind a snapshot of its crashing state. In general, the snapshot of a crashing state is organized in the form of a core dump, which oftentimes contains the crashing program stack, the final values of local and global variables, and the final values of processor registers.

Since a core dump carries certain clues as to a program crash, commercial software vendors oftentimes utilize it to facilitate failure diagnosis and classify crashes likely caused by the same defect [15–17]. For example, Microsoft’s tool RETracer [18] parses a core dump and extracts information such as the crash point and the crashing stack. Then, it employs a backward taint analysis technique to infer program faults and further triages program crashes. While shown to be effective in spotting the function that contributes to a crash, existing technical approaches (e.g., [17, 18]) are less likely
to be effective in identifying some program faults, particularly memory corruption vulnerabilities (e.g., buffer overflow and use after free).

A memory corruption vulnerability is a special type of faults in software that could lead to unintentional modification to the content at a memory location and thus compromise the data dependency of a running program. As such, a core dump may carry a certain amount of corrupted data when a memory corruption vulnerability is triggered and incurs a program crash. Since corrupted data can be anywhere in the memory, it leaves a significant challenge for identifying debugging information. In attempting to exploit a buffer overflow vulnerability, for example, an attacker typically overwrites adjacent memory locations. As we will show later in Section 3.2, this may significantly increase the difficulty in identifying a stack trace and even spotting the crash point. Since a crash point and stack trace are the most useful information for failure diagnosis, without a proper mechanism to locate them in a core dump, a core dump is practically useless.

In fact, existing core dump analysis techniques can barely serve as informative debugging aids in locating a memory corruption vulnerability, even though there is no impediment in tracking down the crash point and stack traces of a crashing program. As is mentioned above, a memory corruption vulnerability allows an attacker to compromise the data dependency of a running program. In facilitating failure diagnosis, existing techniques typically perform backward program analysis starting from the crash point, and assume the integrity of a crashing stack is not compromised (e.g., [18, 102]). When such techniques intersect corrupted data, therefore, they may terminate unexpectedly and produce no information other than the crash point and stack traces of a crashing program.

In this work, we develop CREDAL, an automatic debugging tool to assist software developers in tracking down software faults, particularly memory corruption vulnerabilities. Our goal is not to let CREDAL pinpoint a memory corruption vulnerability, but rather to turn a core dump to an informative aid in locating the vulnerability. To deal with the challenges that memory corruption introduces to core dump analysis, technically, CREDAL leverages the source code of the crashing program to enhance core dump analysis.
Since the state of the program at the crash is an almost-necessary starting point [19] and stack traces can potentially narrow down the list of candidate files that are likely to contain software defects [20], CREDAL first identifies the crash point and attempts to restore the stack trace of a crashing program.

In general, it is easy to identify a crash point. When a program has crashed and terminated unexpectedly, the final value of the program counter typically indicates the crash point. However, memory corruption may manipulate the program counter, making it point to an invalid instruction. (e.g., overwriting a return address on the stack with a non-executable memory location). To address this problem, CREDAL checks the validity of the program counter. For the invalid program counter, CREDAL restores its value by analyzing the remnants on the stack. More specifically, CREDAL first attempts to identify the function which was just called but silently returned before the crash, in that this function carries the information about its parent (i.e., the crash function). Using this function, CREDAL then locates the crash function as well as the crash point within it. In recovering the crashing stack, CREDAL makes conservative inference using the restored program pointer along with call frame information [21].

As is discussed above, memory corruption can manipulate the content at a certain memory location, which may result in the violation of data dependency. Intuition suggests that highlighting data dependency mismatch seems informative for failure diagnosis. As a result, we further augment CREDAL with the ability of specifying data dependency mismatches at the source code level. In particular, CREDAL identifies the variables – the values of which in memory mismatch the data dependency of the crashing program – and highlights the source code corresponding to the mismatches. Technically speaking, CREDAL first constructs an inter-procedural control flow graph based on the stack traces restored as well as the source code of the crashing program. Then, it performs inter-procedural points-to analysis and reaching definition analysis to discover the mismatch in variable values and pinpoint the code fragments corresponding to the mismatch.

We implemented CREDAL for Linux systems on x86 platform. To the best of our
knowledge, CREDAL is the first automatic tool that can perform core dump analysis in the condition where a core dump contains a certain amount of corrupted data. We manually analyzed 80 crashes corresponding to 73 memory corruption vulnerabilities collected from Offensive Security Exploit Database Archive [108], and compared our manual analysis with the analysis conducted by CREDAL. We observed that CREDAL can accurately identify a crash point and (fully or partially) recover stack traces from a core dump. In addition, we demonstrated that CREDAL can potentially increase the utility of a core dump. For about 80% of the crashes, CREDAL can narrow down vulnerability diagnosis within a couple of functions. For about 50% of the crashes, CREDAL can bound diagnosis efforts in only tens of lines of code.

In summary, we make the following contributions.

- We designed CREDAL, an automatic debugging tool that leverages the source code of the crashing program to enhance core dump analysis and provides useful information for software failure diagnosis.
- We implemented CREDAL on Linux for facilitating software developers (or security analysts) to locate software faults, particularly memory corruption vulnerabilities.
- We demonstrated the utility of CREDAL in facilitating memory corruption vulnerability diagnosis by using 80 crashes attributable to 73 memory corruption vulnerabilities.

The rest of this chapter is organized as follows. Section 3.2 defines the problem scope of our research. Section 3.3 presents the overview of CREDAL. Section 3.4 and 3.5 describe the design and implementation of CREDAL in detail. Section 3.6 demonstrates the utility of CREDAL. Finally, we conclude this work in Section 3.8.

### 3.2 Problem Scope

In this section, we define the problem scope of our research. We first discuss our threat model. Then, we demonstrate how a memory corruption vulnerability can undermine the
utility of a core dump with a real world example.

3.2.1 Threat Model

Our research focuses on diagnosing the crash of a process. Therefore, we exclude the program crashes that do not incur the unexpected termination of a running process (e.g., Java program crashes). Because our research diagnoses a process crash through core dump analysis, we further exclude the process crashes that typically do not produce core dumps. Up to and including Linux 2.2, the default action for CPU time limit exceeded, for example, is to terminate the process without a core dump.

As is mentioned above, our research is motivated by memory corruption. As a result, we only deal with process crashes caused by memory corruption vulnerabilities. Although many software defects can trigger a process crash, and CREDAL can provide useful information for diagnosing any process crashes, the software defects that can trigger a program crash but not result in memory corruption are out of our research scope. In general, such defects include buffer over-read, null pointer accesses, uninitialized variables, and out-of-memory errors. We believe this is a realistic threat model because (1) it covers all the memory corruption vulnerabilities and (2) techniques to analyze excluded software defects have been proposed by other researchers and can be combined with CREDAL.

Note that we design CREDAL as a debugging tool to analyze crashes triggered by memory corruption during random exercises. We do not assume CREDAL can act as a defense meant to work in an adversarial setting where the attackers can actively prevent offline debugging.

3.2.2 Motivating Example

We use a real world vulnerability – CVE-2013-2028 [109] – as a typical example to illustrate how and why a memory corruption vulnerability can compromise the integrity
static int ngx_http_read_discarded_request_body(ngx_http_request_t *r){
    size_t size;
    ssize_t n;
    ngx_int_t rc;
    ngx_buf_t b;
    u_char buffer[NGX_HTTP_DISCARD_BUFFER_SIZE];
    ...
    ngx_memzero(&b, sizeof(ngx_buf_t));
    b.temporary = 1;
    for ( ;; ) {
        ...
        // Choose the minimum value between the two arguments
        size = (size_t) ngx_min(r->headers_in.content_length_n,
             NGX_HTTP_DISCARD_BUFFER_SIZE);
        //copy data to buffer
        n = r->connection->recv(r->connection, buffer, size);
        ...
        b.pos = buffer;
        b.last = buffer + n;
        rc = ngx_http_discard_request_body_filter(r, &b);
        if (rc != NGX_OK) {
            return rc;
        }
    }
}

Table 3.1: A code fragment from Nginx-1.4.0 vulnerable to an integer overflow (CVE-2013-2028).

of a program counter and tamper data on the stack, making a core dump futile for software debugging.

Table 3.1 shows a code fragment from Nginx-1.4.0. As is described in the case of CVE-2013-2028, this code fragment can manipulate a signed integer and trigger a stack based overflow. More specifically, an attacker can craft a request and thus manipulate the value of r->headers_in.content_length_n. When handling this specifically crafted request, as is shown in line 15, a worker process compares the value held in r->headers_in.content_length_n with a constant, chooses the
minimum and assigns it to variable size. Then, the worker process uses this variable to
determine the number of bytes it needs to copy from memory area r->connection to
memory area buffer (see line 17).

In this example, r->headers_in.content_length_n is a signed integer
type, whereas variable size is an unsigned type. This inconsistency can potentially
triggers a stack overflow which may further results in a crash and even arbitrary code
execution. Specifically, an attacker can set r->headers_in.content_length_n
to negative. Due to the inconsistency in variable type, the value of r->headers_-in.content_length_n can be misinterpreted as a very large positive number when
casting to unsigned type variable size. As is defined in line 8, array buffer can
only carry NGX_HTTP_DISCARD_BUFFER_SIZE bytes of data. Since the value in
size is larger than constant NGX_HTTP_DISCARD_BUFFER_SIZE, the request can
overflow buffer and corrupt the data on the stack.

Figure 3.1 illustrates part of information in a core dump after exploiting the afore-
mentioned vulnerability with a publicly available PoC [110]. We observe the exploit
overflows buffer and corrupts the local variables, argument and return address of
function ngx_http_read_discarded_request_body. For this specific exam-
ple, these corrupted data on the stack do not interrupt the process execution until the
function returns. At the time the function returns, the running process simply restores
frame pointer EBP to the previous value held in it\(^1\), and sets program counter EIP to the
return address on the stack. As such, we observe both EBP and EIP are set to an invalid
address (0x41414141) at the time of the crash.

Since EBP is designed to provide a frame pointer for the current function, and EIP
holds the address of the next CPU instruction, at the time of a program crash, their
snapshots typically indicate the crash function and crash point, respectively. With both
process registers holding an invalid address at the time of the crash, however, software

\(^1\)Note that most compilers provide an option to omit frame pointers. With that option enabled, it makes
debugging even more difficult. To illustrate the difficulty in identifying debugging information in a core
dump, we assume software developers can retrieve information from EBP to assist their failure diagnosis.
developers receive no clue as to the program crash. In the following sections, we will therefore develop new technical approaches to identify the crash point as well as the stack traces of a crashing program.

### 3.3 Overview

In this section, we discuss our design principle followed by the description about how CREDAL performs core dump analysis at a high level.

**Figure 3.1:** The status of the crashing stack and processor registers after exploiting the overflow vulnerability specified in CVE-2013-2028. For simplicity and demonstration, the stack protector has been disabled.
3.3.1 Design Principle

When locating a software defect, it is always beneficial for software developers to narrow down the manual efforts to code with as few lines as possible. Ideally, we would like to minimize the manual effort of a developer by designing CREDAL to pinpoint a software defect directly. However, a core dump may carry a certain amount of corrupted data, and the information held in it only provides a partial chronology of how the program reached a crash point. To track down a software defect, therefore, we need to design CREDAL to infer the ambiguity about program execution. This potentially increases the uncertainty in the information that CREDAL provides to the developers. Considering such uncertainty may mislead failure diagnosis, our design follows a conservative principle – maximizing the reliability of the information that CREDAL produces by minimizing the uncertainty in core dump analysis.

3.3.2 Technical Approach

In an extreme case, we can design CREDAL to achieve zero uncertainty in core dump analysis by giving no information to software developers. However, such a design sacrifices the utility of a core dump in failure diagnosis. To balance between utility and uncertainty, we therefore design CREDAL as follows.

As mentioned in Section 1, a crash point typically serves as the starting point of failure diagnosis, and a stack trace can narrow down the list of candidate files that possibly contain software defects. As a result, we design CREDAL to provide this essential information needed by software developers and security analysts. Considering memory corruption discards data dependency, and presenting data dependency mismatch may facilitate failure diagnosis, we also design CREDAL to highlight the code fragments corresponding to data dependency mismatch.

To track down the crash point, CREDAL extracts the final value of the program counter stored in a core dump simply because a program counter at the crash reveals where a
program crash occurred. As mentioned in Section 3.2, a memory corruption vulnerability may overwrite a program counter to an invalid value. Before using it to pinpoint the crash point, CREDAL therefore checks the validity of the program counter. For the program counter with an invalid value, CREDAL attempts to restore its value using the data within a previously returned stack frame. We will describe more details in the following section.

To identify a stack trace, CREDAL follows the DWARF standard [21] to unwind a crashing stack. CREDAL can traceback all the functions that have been called but not yet returned at the time of the crash. Note that memory corruption undermines the data on the stack and may thwart stack frame identification. Following the design principle above, CREDAL stops stack frame identification and produces a partial stack trace when identifying a stack frame that does not match certain heuristics. Figure 3.2 shows one situation where a stack trace cannot be fully identified in an accurate manner and CREDAL terminates in advance. The program crashes in function crash(). Using the remnants stored on its stack frame, CREDAL computes the Canonical Frame Address.
(CFA) and tracebacks to parent function \texttt{foo()}. However, a stack overflow occurred in \texttt{foo()}, overwrote the data held in the frame and made it invalid. Considering that the program may have an earlier call to \texttt{bar1()} or \texttt{bar2()}, and there is insufficient information about the execution path in the lead-up to the crash, CREDAL terminates stack trace identification and outputs a partial stack trace with function \texttt{crash()} and its caller \texttt{foo()}.

To pinpoint data dependency mismatch, CREDAL first constructs a control flow graph. Given the graph along with the aforementioned crash point and stack trace, CREDAL further performs an inter-procedural points-to analysis and an inter-procedural reaching definition analysis. These analyses allow CREDAL to obtain a set of data dependency constraints and thus identify the variables with mismatching dependency. For the variables with mismatching values, CREDAL further highlights the code fragments corresponding to the mismatch, and presents the code fragment with minimal number of lines of code to software developers (or security analysts). The intuition here is that these code lines may be potentially used as reference for locating a memory corruption vulnerability. In the following section, we will discuss CREDAL with more technical details.

### 3.4 Design

In this section, we discuss the technical details of CREDAL. Specifically, we start with crash thread identification. Then, we describe how CREDAL identifies the crash point, stack trace and data dependency mismatch in detail. In addition, we specify the uncertainty that CREDAL may introduce in core dump analysis, and discuss how we leverage technical approaches to minimize this uncertainty.

As is mentioned earlier, a core dump carries the values of processor registers and the values stored in memory, which can be directly consumed by binary-level analysis. As such, we perform core dump analysis mainly on binaries. Considering data flow analysis
is typically performed on source code, and source code is self-evident for software developers, we therefore translate the information derived from binary-level analysis into a form that is amenable to source code level analysis. In Section 3.5, we will describe how to implement CREDAL to map the values in memory and x86 instructions to variables and the statements in source code.

3.4.1 Discovering Crashing Thread

A process may contain multiple threads. When it crashes, an operating system includes recorded state of the working memory of each thread in a single core dump. To provide useful, interesting information for crash diagnosis, CREDAL first needs to identify the crashing thread in the core dump.

In this work, CREDAL employs the state of the program counter to identify a crashing thread. In particular, CREDAL examines the program counter of each thread at the crash. When CREDAL discovers a program counter that points to an invalid memory address or an illegal instruction (e.g., the instruction containing an invalid opcode or incurring a floating point exception), CREDAL deems the corresponding thread as the one that crashes the process. For the situation where a program counter points to a valid instruction but the instruction attempts to access an invalid memory address, CREDAL also treats the corresponding thread as the one contributing to the crash.

3.4.2 Tracking down Crash Point

As is mentioned in Section 3.3, a crash point is typically enclosed in the program counter at the crash. Due to the corrupted data on stack and in processor registers, the program counter may hold an invalid value, making crash point identification difficult.

Here, we deal with this technical issue using the previously returned stack frames. The intuition here is that a crash function generally does not overwrite the stack frame allocated for the earlier function call, and the data on this stack frame can facilitate the identification of the crash point. Figure 3.2 illustrates an example where a program unex-
pectedly terminated in function crash() but the data on the stack frame corresponding to an earlier call to bar() has not yet been overwritten. Since the return address of function bar() is stored on its stack frame, and directly points to the next instruction that would be executed in function crash(), the crash function can be easily identified through this linkage.

However, a crashing stack does not provide sufficient information that can help pinpoint previously returned stack frames on a crashing stack. To address this problem, we scan a crashing stack through a sliding window, looking for the return address of the function that was just called (but presumably silently returned) before the crash. The intuition here is that a function pushes the return address of its child on the stack at the time the child is invoked, and we use the child as the indicator of the stack frame corresponding to the function.

In this work, we set the size of the sliding window to memory address width (e.g., 4 bytes for a 32-bit operating system). The scan of the crashing stack starts from the top of the stack indicated by the value of stack pointer ESP plus an offset equal to the memory address width (e.g., ESP+4 for a 32-bit operating system).

CREDAL follows two criteria when determining if the value held in the sliding window represents the return address of the previously returned function. As a return address points to the instruction that would be executed after a function returns, CREDAL first ensures the value in the window links to a valid instruction. Second, CREDAL examines the instruction above the one corresponding to the return address. In particular, CREDAL checks if that instruction is a call instruction because a ret instruction indicates the completion of a function call.

For the value in the sliding window that matches the criteria above, CREDAL deems it as a valid return address candidate, and follows the aforementioned steps to pinpoint the crash function. With the crash function identified, CREDAL further performs program counter recovery. Since memory corruptions manipulate the program counter through indirect jump instructions (e.g., ret; call EDX), and point it to an invalid memory
address, CREDAL examines indirect jump instructions in the crash function.

In particular, CREDAL first uses an intra-procedural Control Flow Graph (CFG) to identify all the indirect jump instructions reachable from the instruction corresponding to the candidate return address. Figure 3.2 shows an intra-procedural CFG for function crash(). In this CFG, we prune all the direct function calls and cut off corresponding connections. The intuition here is that the crashing stack exhibits a different layout if the crashing function calls another subroutine after the instruction corresponding to the return address. Again, take the example code shown in Figure 3.2. The return address on the crashing stack points to an instruction in crash function crash(). The CFG within the crash function indicates a call to function foo2(). If the program invokes function foo2() and crashes after, the stack frame of function bar() would not be presented on the crashing stack.

Second, CREDAL verifies the destinations of indirect jump instructions identified on the aforementioned CFG. More specifically, CREDAL computes the destination of each indirect jump instruction using the values of processor registers or memories preserved in the core dump. Then, CREDAL attempts to match the destination with the value held in the program counter. When identifying a match, CREDAL restores the program counter to the address of that indirect jump instruction and deems it as the crash point. Note that if the information is incomplete for CREDAL to recover the crash point, our analysis terminates.

The aforementioned program counter recovery mechanism follows systematic analysis and verification. However, it still introduces uncertainty to crash point identification. For example, a crash function does not invoke any subroutine before the crash, and CREDAL mistakenly identifies the remnants on the stack as a valid return address. To minimize such uncertainty, CREDAL further verifies the identified crash point by checking the displacement of the stack pointer. Figure 3.3 shows an example where crash function crash() does not invoke any subroutines but the stack frame of an earlier call to bar1() is preserved. Thus, function foo1() is mistakenly identified as the
crash function. By examining the displacement of the stack pointer before and after the call to `bar1()`, however, CREDAL can identify the difference between the expected and actual stack pointer. As is illustrated in Figure 3.3, the position of ESP should be at the bottom and the top of the stack frame of `bar1()` at the entry and exit of the function, respectively. Since there is no other operation to ESP within function `foo1()`, the expected position for ESP should remain at the top of the frame of `bar1()` at the crash. However, this expectation does not match the observation from the core dump, which indicates the incorrectness of crash point identification. The intuition behind this verification is that the stack frame of the crash function mistakenly identified generally does not share the same size as that of the actual crash function.

With the design discussed above, we believe CREDAL can identify a crash point with high confidence because it is very unlikely to bring about the coincidence where (1) the remnants on the crashing stack are mistakenly deemed as an address that points to a valid instruction; (2) the instruction above the valid instruction is a call instruction; (3) an indirect jump in the crash function mistakenly identified encloses a destination that happens to share the same value as the corrupted program counter; (4) the stack frame of the crash function mistakenly identified happens to share the same size as that

Figure 3.3: Stack pointer verification. For simplicity and demonstration, the return address points a line of C programming code converted from the instruction on a binary.
of the actual crash function. In Section 3.6, we will demonstrate the effectiveness and correctness of CREDAL in tracking down a crash point.

### 3.4.3 Identifying Stack Trace

With the crash point identified, we now discuss how to use it to track down a stack trace. As is described in Section 3.3, CREDAL has the access to the source code of a crashing program. Thus, it can compile the code with debugging options enabled, and obtain the call frame information of the crashing program.

The call frame information is typically used for stack unwinding. Following DWARF standard [21], therefore, CREDAL can “virtually” unwind a crashing stack and track down a stack trace. As is discussed in Section 3.3, data corruption on stack however may introduce uncertainty to stack unwinding. As a result, CREDAL also verifies the legitimacy of a stack frame in each step in addition to following the restored registers to find all stack frames.

More specifically, CREDAL walks the crashing stack and checks the validity of the return address in each stack frame by following the criteria discussed in Section 3.4.2. In addition, CREDAL examines the allocation of a newly unwound stack frame and ensures it is laid just on top of the last stack frame successfully identified. The intuition here is that the frames on stack should be compactly laid out but not overlapped. Similar to the approach we leverage in Section 3.4.2, CREDAL finally verifies the size of a newly identified stack frame using the displacement of the stack pointer.

As is discussed in the previous section, the design of CREDAL follows a conservative principle. When “virtually” unwinding a crashing stack and identifying a stack frame cannot pass the aforementioned verification, CREDAL stops the unwinding operation and produces a partial stack only with the stack frames successfully identified. We design CREDAL to conservatively identify stack trace so that corrupted stacks can also be handled. In Section 3.6, we will demonstrate the correctness of CREDAL in partially (or fully) identifying a stack trace.
Figure 3.4: An inter-procedural control flow graph and data dependency constraints. Note that `scanf()` is a call to an external library. For simplicity and demonstration, we do not unfold this call. The list of constraints defines data dependency e.g., $a_{@\text{foo}}=1$ indicates variable $a$ in function $\text{foo}$ should be equal to 1 at the crash; $d_{@\text{foo}}=[\text{valid}]$ indicates pointer $d$ should be valid at the crash.

### 3.4.4 Discovering Data Dependency Mismatch

As is mentioned above, memory corruption typically incurs data corruption. If the value of a variable observed in the core dump does not match any reachable definition, a data dependency mismatch is found. Here, we describe how CREDAL pinpoints such a dependency mismatch and highlights the corresponding code fragment.

On a high level, we statically analyze the set of possible values for each variable on the recovered trace and match the possible values with the actual value in the core dump. Assuming our analysis on possible value sets is sound, if the value of a variable indicated by the core dump falls out of the corresponding value set, a memory corruption must have occurred.

To obtain such value sets, we perform an inter-procedural reaching definition analysis with the restored stack trace. As we will describe in the following presentation, our analysis is conservatively designed to avoid loss of soundness. Specifically, our analysis firstly constructs an inter-procedural CFG that covers all possible call sequences from the entry function to the crash point according to the recovered stack. As resolving indirect calls may introduce inaccuracy, CREDAL skips indirectly called functions (whose targets are unknown) but preserves their arguments. Although skipping indirect calls results in a
partial CFG, our subsequent analysis will conservatively consider the potential effects of these indirectly called functions, to make the whole analysis sound. Figure 3.4 illustrates the CFG corresponding to the example in Figure 3.2.

To the partial CFG, we apply an intra-procedural points-to analysis to each function, following a context and path insensitive strategy. With the points-to information, we can easily calculate the reaching definition in each function. We then populate the intra-procedural results across function boundaries and extend the results to the whole CFG. We achieve this using a summary-based inter-procedural static analysis algorithm (i.e., the "functional approach") [111]. More specifically, we capture the effects caused by a function modifying variables in another function through pointers passed as arguments. To guarantee the soundness of our results, we handle indirectly called functions in a conservative manner. To be specific, we assume that an indirect call modifies all global variables and all variables possibly pointed to by the argument. We assume those variables may equal any value after the indirect call. Take vulnerable code in Figure 3.2 for example, our analysis gets rid of global variable glob and local variable b when analysis reaches to line 46, since glob_func represents an indirect call which receives pointer c to local variable b.

After obtaining the reaching definition results, we deduce the possible value set for each variable (i.e., value constraints on each variable). If one definition can be tracked back to a constant value, we add the constant to the set. Otherwise, we assume the definition leads to all possible values. Afterwards we start searching for corrupted variables in the core dump. Note that CREDAL does not consider global variables if the crashing program has multi-threading, because global variables are shared by all threads and a core dump does not unveil when a global variable was modified by other threads. We also do not consider those variables if the crashing stack does not preserve their values (e.g., variable x in function sub()). The intuition here is that there is no sufficient evidence to examine data dependency mismatch if the final value of the variable is unknown.
For a variable of non-pointer type, if its final value in the core dump does not match the value constraints, we determine a dependency mismatch with this variable. Further, we highlight the code fragment from all the reachable definitions of this variable to the crash point in the CFG. Take variable \(a\) in Figure 3.2 for an example. \(a\) has one definition, namely \(a = 1\) at line 47. The final value of \(a\) is corrupted and deviates from 1, thus we catch a mismatch on \(a\). The code segment is determined as from line 47 to line 50 and line 55 to line 62.

For a variable of pointer type, CREDAL first takes it as a normal non-pointer variable and performs the above check. In addition, CREDAL searches for another type of dependency mismatch. Assuming on any path from the first function to the crash point on the CFG, there exists at least one dereference to this pointer without subsequent re-definition, the pointer must have been unintentionally manipulated. The intuition is that if there is no unintended manipulation, the process should have crashed in the previous dereferences. CREDAL deems this as dependency mismatch on pointer dereference. Similarly, CREDAL highlights the code fragment from all the dereferences to the crash point in the CFG.

### 3.5 Implementation

We have implemented a prototype of CREDAL for Linux 32-bit system, which takes as input a core dump file as well as the binary and source code of the corresponding crashing program. As CREDAL needs debugging information for analysis, our implementation requires the binary to be compiled with debugging options (e.g., `-g` with `gcc`). In this section, we present some important implementation details.

Linux operating system organizes a core dump file in the form of Executable Linkable Format (ELF). The implementation of CREDAL employs libelf, an open source library [112] to parse the file in ELF and retrieve the corresponding memory information. Considering CREDAL needs to examine the entire working memory of a crashing pro-
gram, and Linux kernel typically does not include file-backed mappings in a core dump, our implementation augments libelf with the ability to interpret the note segment in an ELF file so that CREDAL can identify file-backed memory mappings and consume the information in that memory area.

CREDAL currently relies on the debug information to disassemble binaries and unwind crashing stacks. For disassembly, our implementation uses libdwarf library [113] to parse a binary and then employs libdisasm library [114] to identify instructions in it. For unwinding a crashing stack, our implementation relies on libdwarf library to extract call frame information from .debug_frame and .eh_frame stored in ELF files. To perform virtual unwinding, we also implement CREDAL by modifying libunwind library [115].

As is mentioned in Section 3.4, CREDAL constructs an intra-procedural CFG on a binary. However, indirect jump instructions introduce non-deterministic to the CFG construction. Given instruction [jmp %EAX], for example, it is difficult to construct the consecutive nodes on CFG without determining the destination of this instruction. To address this problem, our implementation uses LLVM [116] to extract program semantics from source code and identify the destinations of indirect jump instructions. For example, assume an indirect jump instruction is a low-level representation of a switch statement in C programming language. Our implementation employs LLVM APIs to identify the destinations from those case statements, and completes CFG construction.

In our design, CREDAL utilizes the displacement of a stack pointer to verify the crash point and stack trace identified. To do this, CREDAL needs to know the change to ESP for a given code fragment. Within the code fragment, there may be a variety of execution paths that cover the operations of ESP (e.g., [add $0x4, %ESP], [pop] or [push %EAX]). Theoretically, stack pointer ESP may end up at different position when going through different paths. To guarantee the correctness of program execution, a compiler however ensures ESP has the same displacement whichever paths the program walks through. As a result, our implementation chooses an arbitrary path to compute the
displacement of ESP when verifying a crash point or stack trace.

Our implementation seeks for data dependency mismatches on LLVM IR. We construct the call graph in an on-demand manner. We compile each source code file separately into LLVM IR and traverse from the first function on the recovered stack trace to the crash point. Whenever a function is directly called, we include that function and expand the call graph. If that function is in a different IR file, we search for it in other compile units by its name and linkage type. With the call graph and the intra-procedure CFG natively provided by LLVM, we essentially have the inter-procedure CFG.

To the CFG, we apply LLVM’s built-in basic alias analysis to get intra-procedure point-to results. To ensure the soundness of the analysis, our implementation takes may-alias relation as must-alias relation. Though the conservativeness theoretically limits the analysis capability of CREDAL, our case studies in Section 3.6 demonstrate that this implementation can work well in practice. LLVM also provides intra-procedural reaching definition results through its use-def chain data structures. With these two pieces of information, we further implemented a simple summary-based inter-procedural analysis as described in Section 3.4.

As is discussed in Section 3.4, CREDAL examines variables in LLVM IR, the values of which are preserved in the core dump. Our implementation utilizes debugging information in the IR to achieve mapping between source code variables and LLVM IR variables and we again leverages debugging information in the binary to compute the locations of source code variables in the core dump. We highlight the code segment for data dependency mismatch in LLVM IR, which is also mapped to source code segment via debugging information in the IR file.

### 3.6 Case Study

In this section, we demonstrate the utility of CREDAL using the crashes attributable to memory corruption vulnerabilities. In particular, we describe the collection of crashes
and present the effectiveness of CREDAL. We also discuss those memory corruption vulnerabilities, the crashes of which CREDAL fails to handle.

### 3.6.1 Setup

To demonstrate the utility of CREDAL, we must collect program crashes contributed by memory corruption vulnerabilities. We exhaustively searched memory corruption vulnerabilities on Offensive Security Exploit Database Archive [108]. Our goal is to gather the crashes by exploiting memory corruption vulnerabilities with corresponding PoCs.

As an outdated vulnerability typically associates with an obsolete program, and such a program may be no longer available, we only gathered memory corruption vulnerabilities archived over the past twelve years. Because we implement CREDAL for Linux operating system, we further narrowed down our searches on the vulnerabilities identified on software running on Linux. In total, we obtained 392 memory corruption vulnerabilities bundled with at least one PoC. We compiled and configured vulnerable programs based on the description of the collected vulnerabilities, and successfully produce 80 crashes using the PoCs corresponding to 73 vulnerabilities. The experiments are conducted on a machine with Intel Xeon E5-2560 2.30GHz and 2GB memory running Ubuntu 14.04. The average time to analyze a core dump is 0.21 seconds.

Table 3.2 lists the aforementioned crashes and summarizes the corresponding vulnerabilities across 5 different categories, including use-after-free as well as overflow on stack, heap, integer and bss/data. These vulnerabilities are identified on 62 distinct software, ranging from sophisticated programs like PHP and Binutils with lines of code over 670K to lightweight programs such as o3read and corehttp with lines of code less than one thousand.

Note that we discard a large fraction of vulnerabilities for three reasons. First, the program corresponding to a vulnerability is obsolete and we cannot find its source code for showcasing the utility of CREDAL (e.g., Gaim [117] and Abc2midi [118]).
<table>
<thead>
<tr>
<th>Program</th>
<th>Program Size (LOC)</th>
<th>CVE-ID</th>
<th>Vulnerability Type</th>
<th>EDB-ID</th>
<th>EIP # of Frame</th>
<th>Full Stack</th>
<th>Area (LOC)</th>
<th>Root Cause</th>
<th># of Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>tack 1.0.7</td>
<td>20979 NA</td>
<td>BSS/Data Overflow</td>
<td>38685</td>
<td>2</td>
<td>1</td>
<td>27</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>picpuz 2.1.1</td>
<td>6345 NA</td>
<td>BSS/Data Overflow</td>
<td>10634</td>
<td>2</td>
<td>25</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ffmpeg 3</td>
<td>2644 NA</td>
<td>BSS/Data Overflow</td>
<td>24984</td>
<td>4</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>yterm 1.94</td>
<td>14010 NA</td>
<td>BSS/Data Overflow</td>
<td>39406</td>
<td>5</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nullhptd 0.5.0</td>
<td>1849 NA</td>
<td>Heap Overflow</td>
<td>218181</td>
<td>6</td>
<td>X</td>
<td>15</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>make 3.8.1</td>
<td>24151 NA</td>
<td>Heap Overflow</td>
<td>34164</td>
<td>1</td>
<td>X</td>
<td>12</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xrdp 0.4.1</td>
<td>33995 2008-5904</td>
<td>Heap Overflow</td>
<td>8469</td>
<td>14</td>
<td>X</td>
<td>224</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>libsndfile 1.0.25</td>
<td>51064 2015-7805</td>
<td>Heap Overflow</td>
<td>38447</td>
<td>26</td>
<td>X</td>
<td>280</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ntpd 4.2.6</td>
<td>152433 NA</td>
<td>Heap Overflow</td>
<td>39445</td>
<td>X</td>
<td>4</td>
<td>33</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>php 5.3.6</td>
<td>805640 2012-2386</td>
<td>Heap Overflow</td>
<td>17201</td>
<td>10</td>
<td>X</td>
<td>NA</td>
<td>NA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>inetutils 1.8</td>
<td>98941 NA</td>
<td>Heap Overflow</td>
<td>15705</td>
<td>3</td>
<td>X</td>
<td>NA</td>
<td>NA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nginx 1.4.0</td>
<td>100255 2013-2028</td>
<td>Integer Overflow</td>
<td>25775</td>
<td>1</td>
<td>X</td>
<td>36</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Overkill 0.16</td>
<td>16361 2006-2971</td>
<td>Integer Overflow</td>
<td>1894</td>
<td>5</td>
<td>X</td>
<td>4</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>php 5.16</td>
<td>561820 2007-1001</td>
<td>Integer Overflow</td>
<td>29823</td>
<td>14</td>
<td>X</td>
<td>46</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ImageMagick 6.2.0</td>
<td>214800 2006-4144</td>
<td>Integer Overflow</td>
<td>28283</td>
<td>5</td>
<td>X</td>
<td>110</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>php 4.4.4</td>
<td>367295 2007-1777</td>
<td>Integer Overflow</td>
<td>29788</td>
<td>6</td>
<td>X</td>
<td>NA</td>
<td>NA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>python 2.2</td>
<td>416060 2007-4965</td>
<td>Integer Overflow</td>
<td>30592</td>
<td>11</td>
<td>X</td>
<td>NA</td>
<td>NA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ClamAV 0.88.2</td>
<td>42160 2006-4182</td>
<td>Integer Overflow</td>
<td>2587</td>
<td>1</td>
<td>X</td>
<td>NA</td>
<td>NA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mcrypt 2.5.8</td>
<td>10363 2012-4409</td>
<td>Stack Overflow</td>
<td>22938</td>
<td>1</td>
<td>X</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>putty 0.66</td>
<td>90165 2016-2563</td>
<td>Stack Overflow</td>
<td>39551</td>
<td>1</td>
<td>X</td>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>php 5.2.4</td>
<td>803486 2011-1938</td>
<td>Stack Overflow</td>
<td>17318</td>
<td>1</td>
<td>X</td>
<td>43</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rsync 2.5.7</td>
<td>19487 2004-2093</td>
<td>Stack Overflow</td>
<td>152</td>
<td>X</td>
<td>1</td>
<td>27</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>corehttp 0.5.3a</td>
<td>935 2007-4060</td>
<td>Stack Overflow</td>
<td>4243</td>
<td>2</td>
<td>X</td>
<td>6</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>aireplay-ng 1.2-beta3</td>
<td>62656 2014-8322</td>
<td>Stack Overflow</td>
<td>35018</td>
<td>X</td>
<td>1</td>
<td>124</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No-IP DUC 2.19-1</td>
<td>2578 NA</td>
<td>Stack Overflow</td>
<td>25411</td>
<td>X</td>
<td>1</td>
<td>87</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>opendchub 0.8.1</td>
<td>11021 2010-1147</td>
<td>Stack Overflow</td>
<td>11986</td>
<td>X</td>
<td>1</td>
<td>22</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unrr 3.9.3</td>
<td>17575 NA</td>
<td>Stack Overflow</td>
<td>17611</td>
<td>X</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>peercast 0.1214</td>
<td>17193 2006-1148</td>
<td>Stack Overflow</td>
<td>1574</td>
<td>X</td>
<td>52</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>peercast 0.1214</td>
<td>17193 2006-1148</td>
<td>Stack Overflow</td>
<td>1578</td>
<td>X</td>
<td>52</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>peercast 0.1214</td>
<td>17193 2006-1148</td>
<td>Stack Overflow</td>
<td>16855</td>
<td>X</td>
<td>52</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>peercast 0.1214</td>
<td>17193 2007-6454</td>
<td>Stack Overflow</td>
<td>30894</td>
<td>9</td>
<td>X</td>
<td>261</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ettercap 0.7.5.1</td>
<td>53175 2013-0722</td>
<td>Stack Overflow</td>
<td>23945</td>
<td>2</td>
<td>X</td>
<td>32</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>prolzila 1.3.6</td>
<td>13070 2004-1120</td>
<td>Stack Overflow</td>
<td>652</td>
<td>2</td>
<td>X</td>
<td>17</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>prozila 1.3.6</td>
<td>13070 2004-1120</td>
<td>Stack Overflow</td>
<td>1238</td>
<td>X</td>
<td>35</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sftp 1.1.0</td>
<td>1559 NA</td>
<td>Stack Overflow</td>
<td>9264</td>
<td>X</td>
<td>102</td>
<td>X</td>
<td>27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fbxz 2.5.0</td>
<td>15341 NA</td>
<td>Stack Overflow</td>
<td>38681</td>
<td>5</td>
<td>X</td>
<td>144</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conquest 8.2</td>
<td>105784 2007-1371</td>
<td>Stack Overflow</td>
<td>29717</td>
<td>X</td>
<td>54</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tiffsplit 3.8.2</td>
<td>47769 2006-2656</td>
<td>Stack Overflow</td>
<td>1831</td>
<td>2</td>
<td>X</td>
<td>21</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>alsaplayer 0.99.76</td>
<td>26834 2007-5301</td>
<td>Stack Overflow</td>
<td>5424</td>
<td>3</td>
<td>X</td>
<td>13</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xmp 2.5.1</td>
<td>45404 2007-6731</td>
<td>Stack Overflow</td>
<td>30942</td>
<td>X</td>
<td>4</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>proftpd 1.3.0a</td>
<td>111839 2006-6563</td>
<td>Stack Overflow</td>
<td>3730</td>
<td>X</td>
<td>79</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>proftpd 1.3.0a</td>
<td>111839 2006-6563</td>
<td>Stack Overflow</td>
<td>2928</td>
<td>X</td>
<td>79</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>proftpd 1.3.0a</td>
<td>111839 2006-6563</td>
<td>Stack Overflow</td>
<td>3330</td>
<td>X</td>
<td>79</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>proftpd 1.3.0a</td>
<td>111839 2006-6563</td>
<td>Stack Overflow</td>
<td>3333</td>
<td>X</td>
<td>79</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vf 4.1</td>
<td>18734 NA</td>
<td>Stack Overflow</td>
<td>35450</td>
<td>3</td>
<td>X</td>
<td>85</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LibSMI 0.4.8</td>
<td>80461 2010-2891</td>
<td>Stack Overflow</td>
<td>15293</td>
<td>3</td>
<td>X</td>
<td>23</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FENIX 0.92</td>
<td>25236 NA</td>
<td>Stack Overflow</td>
<td>37987</td>
<td>3</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>glib2pzig 2.5.2</td>
<td>1331 2009-5018</td>
<td>Stack Overflow</td>
<td>34356</td>
<td>1</td>
<td>X</td>
<td>7</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hexchat 2.10.0</td>
<td>68181 2016-2233</td>
<td>Stack Overflow</td>
<td>39657</td>
<td>1</td>
<td>X</td>
<td>80</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binutils 2.15</td>
<td>697354 2006-2362</td>
<td>Stack Overflow</td>
<td>27856</td>
<td>4</td>
<td>X</td>
<td>44</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>alsaplayer 0.99.76</td>
<td>26834 2007-5301</td>
<td>Stack Overflow</td>
<td>5424</td>
<td>3</td>
<td>X</td>
<td>13</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>glibc 2.12.90</td>
<td>843348 2015-7547</td>
<td>Stack Overflow</td>
<td>39454</td>
<td>4</td>
<td>X</td>
<td>230</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>gas 2.12</td>
<td>595504 2005-4807</td>
<td>Stack Overflow</td>
<td>28397</td>
<td>1</td>
<td>X</td>
<td>4</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nasm 0.98.38</td>
<td>33553 2004-1287</td>
<td>Stack Overflow</td>
<td>25005</td>
<td>X</td>
<td>1</td>
<td>12</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ringtonetools 2.22</td>
<td>6507 2004-1292</td>
<td>Stack Overflow</td>
<td>25015</td>
<td>2</td>
<td>X</td>
<td>6</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>abcl2tmex 1.6.1</td>
<td>4052 NA</td>
<td>Stack Overflow</td>
<td>25018</td>
<td>1</td>
<td>X</td>
<td>7</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JPeqToAvi 1.5</td>
<td>580 NA</td>
<td>Stack Overflow</td>
<td>24981</td>
<td>3</td>
<td>X</td>
<td>17</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 3.2: The list of the program crashes corresponding to memory corruption vulnerabilities. CVE-ID and EDB-ID specify the IDs of the CVE and corresponding PoC, respectively. EIP indicates the validity of the program counter at the crash. # of Frame and # of Functions describe the number of stack frames CREDAL identifies as well as the number of functions one needs to examine when locating the corresponding vulnerability. The numbers in Area (LOC) indicate the lines of code corresponding to data dependency mismatch.

<table>
<thead>
<tr>
<th>Program</th>
<th>Program Size (LOC)</th>
<th>CVE-ID</th>
<th>Vulnerability Type</th>
<th>EDB-ID</th>
<th>EIP</th>
<th>Full Stack</th>
<th>Area (LOC)</th>
<th>Root Cause</th>
<th># of Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>O3read 0.03</td>
<td>932</td>
<td>2004-1288</td>
<td>Stack Overflow</td>
<td>25010</td>
<td>√</td>
<td>5</td>
<td>7</td>
<td>X</td>
<td>2</td>
</tr>
<tr>
<td>Latex2tf 1.9.15</td>
<td>14473</td>
<td>2004-2167</td>
<td>Stack Overflow</td>
<td>24622</td>
<td>√</td>
<td>2</td>
<td>X</td>
<td>1</td>
<td>√</td>
</tr>
<tr>
<td>libpng 1.2.5</td>
<td>33681</td>
<td>2004-0597</td>
<td>Stack Overflow</td>
<td>389</td>
<td>√</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>unif 0.19.3</td>
<td>5039</td>
<td>NA</td>
<td>Stack Overflow</td>
<td>25303</td>
<td>√</td>
<td>7</td>
<td>√</td>
<td>7</td>
<td>√</td>
</tr>
<tr>
<td>Sox 12.17.4</td>
<td>25736</td>
<td>2004-0557</td>
<td>Stack Overflow</td>
<td>369</td>
<td>√</td>
<td>4</td>
<td>√</td>
<td>83</td>
<td>√</td>
</tr>
<tr>
<td>Sox 12.17.4</td>
<td>25736</td>
<td>2004-0557</td>
<td>Stack Overflow</td>
<td>374</td>
<td>√</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>pspell-p17</td>
<td>1736</td>
<td>NA</td>
<td>Stack Overflow</td>
<td>890</td>
<td>√</td>
<td>2</td>
<td>X</td>
<td>1</td>
<td>√</td>
</tr>
<tr>
<td>streamripper 1.61.25</td>
<td>27304</td>
<td>2006-3124</td>
<td>Stack Overflow</td>
<td>2274</td>
<td>√</td>
<td>3</td>
<td>X</td>
<td>7</td>
<td>X</td>
</tr>
<tr>
<td>Newspost 2.1</td>
<td>4865</td>
<td>2005-0101</td>
<td>Stack Overflow</td>
<td>25077</td>
<td>√</td>
<td>3</td>
<td>√</td>
<td>7</td>
<td>X</td>
</tr>
<tr>
<td>Unalz 0.52</td>
<td>8546</td>
<td>2005-3862</td>
<td>Stack Overflow</td>
<td>26601</td>
<td>√</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>√</td>
</tr>
<tr>
<td>proftpd 1.3.0a</td>
<td>72233</td>
<td>2006-5816</td>
<td>Stack Overflow</td>
<td>2856</td>
<td>√</td>
<td>1</td>
<td>X</td>
<td>36</td>
<td>√</td>
</tr>
<tr>
<td>proftpd 1.3.3a</td>
<td>111839</td>
<td>2010-4221</td>
<td>Stack Overflow</td>
<td>16878</td>
<td>√</td>
<td>1</td>
<td>X</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>ht-editor 2.0.18</td>
<td>119236</td>
<td>NA</td>
<td>Stack Overflow</td>
<td>17083</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Overkill 0.16</td>
<td>16361</td>
<td>2004-0238</td>
<td>Stack Overflow</td>
<td>23634</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>ht-editor 2.0.20</td>
<td>119688</td>
<td>NA</td>
<td>Stack Overflow</td>
<td>22683</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>coreutils 8.4</td>
<td>138135</td>
<td>2013-0221</td>
<td>Stack Overflow</td>
<td>38232</td>
<td>√</td>
<td>5</td>
<td>X</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>mutt 1.4.2.2</td>
<td>61971</td>
<td>2007-2683</td>
<td>Stack Overflow</td>
<td>30093</td>
<td>√</td>
<td>1</td>
<td>X</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>fontforge 201000501</td>
<td>551083</td>
<td>2010-4259</td>
<td>Stack Overflow</td>
<td>15732</td>
<td>√</td>
<td>1</td>
<td>X</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>compface 1.5.2</td>
<td>1574</td>
<td>2009-2286</td>
<td>Stack Overflow</td>
<td>8982</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>openslesse 1.3.19</td>
<td>97241</td>
<td>NA</td>
<td>Use After Free</td>
<td>37051</td>
<td>√</td>
<td>15</td>
<td>√</td>
<td>26</td>
<td>√</td>
</tr>
<tr>
<td>lighttpd 1.4.15</td>
<td>38102</td>
<td>2007-3947</td>
<td>Use After Free</td>
<td>30322</td>
<td>√</td>
<td>6</td>
<td>√</td>
<td>456</td>
<td>X</td>
</tr>
<tr>
<td>python 2.7</td>
<td>100975</td>
<td>2009-2286</td>
<td>Use After Free</td>
<td>100975</td>
<td>√</td>
<td>16</td>
<td>√</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Second, compiling a vulnerable program requires an obsolete external library that we cannot discover (e.g., Asterisk [119] and Blender [120]). Third, a vulnerable program is close-source (e.g., Apple Quicktime [121], Sun Java Runtime Environment [122] and Safari [123]).

3.6.2 Results

To demonstrate the utility of CREDAL, we manually analyze the crashes shown in Table 3.2, and compare our manual analysis with that of CREDAL. More specifically, we evaluate the utility of CREDAL as follow. First, we verify if CREDAL can restore
a program counter and correctly identify the crash point when that is overwritten and
set to an invalid value. Second, we examine if CREDAL can unwind a crashing stack
and pinpoint a full (or partial) stack trace in an accurate manner. Last but not least, we
investigate how effective CREDAL can enclose a memory corruption vulnerability within
the functions and code fragment that it highlights.

3.6.2.1 Pinpointing Crash Point & Stack Trace

Table 3.2 specifies the validity of a program counter at the time of the crash. We observe
21 crashes for which the core dumps carry a program counter with an invalid value.
Among these crashes, CREDAL is able to restore program counters for 20 crashes, and
the program counters recovered all point to crash points correctly.

Table 3.2 also indicates the quantity of the stack frames that CREDAL identifies.
We discovered that CREDAL can fully (or partially) unwind a crashing stack when a
-crash point is successfully pinpointed. The reason is that the crash point reveals the
-crash function in a binary, and even in the worst case, CREDAL can leverage debugging
information to identify the stack frame of the crash function.

We examined the crash for which CREDAL fails to restore the program counter,
particularly the crash of vulnerable program compface 1.5.2. In the actual crash
function, we observed that the function employs setjmp() to save its calling environ-
ment before transferring its execution to a subroutine. The subroutine contains a stack
overflow vulnerability. When exploited, it overflows the current stack frame as well as
those at the higher memory address. The overflow does not block the program execution
immediately. Instead, the subroutine invokes longjmp() which transfers its execution
to a predetermined location in the crash function. In this case, the instruction at the
predetermined location causes an unexpected crash because of the data corruption on the
stack.

Performing analysis for this crash, CREDAL can only discover the stack frame of
longjmp() from the remnants in the core dump. Recall that CREDAL identifies a crash
point using the function that was just called but silently returned before the crash. In this case, the execution of the crash function is not returned from its child function but a descendant function – `longjmp()`. As such, the displacement verification of a stack pointer fails and CREDAL conservatively produces no information about the crash point.

### 3.6.2.2 Locating Vulnerability

In Table 3.2, we also show the lines of code that CREDAL highlights corresponding to data dependency mismatch. For 63 crashes, CREDAL successfully identifies dependency mismatch in memory. Among these crashes, we observe 47 crashes for which CREDAL can enclose the memory corruption vulnerability (i.e., root cause) within the code fragment highlighted. As is shown in Table 3.2, a code fragment highlighted typically covers the statements with only tens of lines (in about 90% cases). This indicates CREDAL has a high potential to reduce manual efforts for locating a memory corruption vulnerability in a crash.

Within the batch of the crashes shown in Table 3.2, there are 16 crashes for which CREDAL identifies dependency mismatch but not encloses the root cause within the code fragment highlighted. For these crashes, we manually examined the code fragment highlighted and the function calls it encloses by imitating the way a security analyst locates a vulnerability. In particular, we started from the code fragment and traversed the enclosed calls in a breadth-first manner. Except for integer overflow in `nginx 1.4.0` and `overkill 0.1.6`, we successfully identified all vulnerabilities in the enclosed function calls. Table 3.2 specifies the number of functions that we walked through when locating a vulnerability. We observe the numbers of the functions we looked into are relatively small, with an average of 3.46. Again, this indicates CREDAL is potentially effective in locating memory corruption vulnerabilities.

In general, overflowing an integer variable does not directly corrupt data in the function where it is enclosed. Rather, it indirectly incurs a buffer overflow and data corruption in a descendant function. For the aforementioned integer overflow, we therefore discov-
ered the overflow vulnerabilities lie outside the code fragment that CREDAL highlights. However, this does not mean CREDAL is less effective in helping security analysts locate integer overflow vulnerabilities. Considering CREDAL typically encloses overflowed buffers in the code fragment, we therefore believe a security analyst can quickly track down integer overflow using the linkage between the overflowed integer variable and the overflowed buffer.

Last but not least, we also manually examined the remaining crashes for which CREDAL fails to identify data dependency mismatch. Except for the one that CREDAL fails to restore the program counter, Table 3.2 specifies 17 crashes in this category. For 9 of them, the failure of CREDAL results from the conservative design of identifying data dependency mismatch.

For 6 of the crashes, the failure of CREDAL can be attributed to one of the following. First, data corruption occurs in the stack area that CREDAL cannot unwind (e.g., ht-editor 2.0.18 & 2.0.20). Second, the corrupted data was sited in local variables that were overwritten by variable assignment operations in consecutive execution (e.g., proftpd 1.3.3 a and vfu 4.1). Third, data corruption occurs at a certain memory area in which the corrupted data has not yet been initialized before the crash (e.g., 2Fax 3 and ytree 1.94).

For the remaining 2 crashes that CREDAL fails to find dependency mismatch, the overflow corresponding to the crashes represents two special cases. In one case, a PoC exploits vulnerable program mutt 1.4.2.2 and underflows the data on stack. At the crash, the program counter points to a call to memmove(). As CREDAL lacks sufficient information to unwind the stack at the higher frame level, and function memmove() does not carry local variables, CREDAL produces no dependency constraint and thus provides less information for locating the overflow vulnerability. In another case corresponding to clamv 0.88.2, CREDAL fails to identify data corruption simply because the crash occurs in advance of data corruption. More specifically, the exploit attempts to overflow a buffer by copying a large data chunk from an invalid memory address.
3.7 Discussion

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

**Other crashes.** CREDAL is designed for providing useful information for software developers (or security analysts) to diagnose the crashes caused by memory corruption vulnerabilities. However, it is not limited to analyzing the crashes that contain data corruption. For the crashes without data corruption, CREDAL only pinpoints the crash point and full stack trace of the crashing program. While such information may not help developers narrow down their debugging efforts within a couple of lines of code, CREDAL still improves the utility of a core dump, especially considering the situation where the program counter points to an invalid address and existing techniques fail to recover it.

**Multiple threads.** CREDAL only focuses on analyzing the data of the crash thread and providing information for debugging. However, a program crash may be contributed by multiple threads. Thus, the information from the crash thread may not help software developers downsize the code space that they have to manually analyze. While this multi-thread issue indeed limits the capability of a security analyst utilizing CREDAL to track down a security vulnerability, this does not significantly downgrade the utility of CREDAL. In fact, a prior study [20] has already indicated that a large fraction of software defects involves only the crash thread. This finding is consistent with our observation from the Offensive Security Exploit Database archive. Looking into the aforementioned vulnerabilities over the past twelve years, we do not discover any vulnerability, the crash of which needs multi-thread coordination.

**Potential attacks.** When demonstrating the utility of CREDAL, we conducted an exhaustive search to find all the PoCs we can experiment with. These PoCs are, however, unaware of CREDAL. Real-world attackers who know about CREDAL might actively...
prevent our analysis. For instance, they may thwart crash point recovery and stack trace recovery via erasing the whole stack, and they may also carefully set up memories to avoid data dependency mismatch. We will take it as our future work to study the possibility of counteracting offline debugging.

### 3.8 Conclusion

In this paper, we develop a debugging tool **CREDAL** to facilitate core dump analysis. With the support from source code, we show that **CREDAL** can enhance core dump analysis and make a core dump more informative for diagnosing software defects, particularly locating memory corruption vulnerabilities. The design of **CREDAL** follows a conservative principle. Thus, it preserves the utility of a core dump, and at the same time, minimizes the uncertainty in core dump analysis.

We demonstrated the utility of **CREDAL** using the crashes corresponding to 73 memory corruption vulnerabilities. We showed that **CREDAL** can accurately pinpoint a crash point as well as a stack trace. In addition, we demonstrated a memory corruption vulnerability typically lies in the code fragment relevant to data corruption. Following this finding, we safely conclude **CREDAL** can significantly downsize the code space that a software developer (or security analyst) needs to manually examine, especially when memory corruption occurs.
4.1 Introduction and Background

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., [124–126]) and core dump analysis (e.g., [18, 19, 127]) 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 inter-leavings 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 [23] – 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 4.2, a post-crash artifact 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

---

1By 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.
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 [128] 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 [128] 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 \texttt{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 \texttt{POMP} is exclusively on the basis of the information residing in post-crash artifacts. In particular, \texttt{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, \texttt{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 [18, 24] or heavily relies upon the capability of recording critical objects
in memory [25–28]. 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 4.4.

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 4.6.

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 4.2 defines the problem scope of our research. Section 4.3 presents the overview of POMP. Section 4.4 and 4.5 describe
void test(void){
    ...
}

int child(int *a){
    a[0] = 1; // assigning value to var
    a[1] = 2; // overflow func
    return 0;
}

int main(){
    void ( *func)();
    int var;
    func = test;
    child(&var);
    func(); // crash site
}

Table 4.1: A toy example with a stack overflow defect.

the design and implementation of POMP in detail. Section 4.6 demonstrates the utility of POMP followed by some discussion on POMP in Section 4.7. Finally, we conclude this work in Section 4.8.

4.2 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.

4.2.1 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 \[129\].

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 \([130–132]\) 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.

### 4.2.2 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.

\(^{2}\)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.
Figure 4.1: 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.

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 4.1. 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 4.1. 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 4.1, 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 4.1). 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 flows, this is not sufficient for completing program failure diagnosis. Again, take for example the execution trace shown in Figure 4.1. 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+0×8] shown in instruction A12. However, memory indicated by [ebp+0×8] 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.

4.3 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.

4.3.1 Intel Processor Tracing

Intel PT is a low-overhead hardware feature available in recent Intel processors (e.g., Skylake series). It works by capturing information pertaining to software execution. To minimize the storage cost, Intel PT organizes the information captured in different forms
of data packets. Of all the data packets, Taken Not-Taken (TNT) and Target IP (TIP) packets are the ones most commonly adopted. Technically speaking, TNT packets take the responsibility of recording the selection of conditional branches, whereas TIP packets are used for tracking down indirect branches and function returns. Along with some other packets such as PGE and PGE, Intel PT also utilizes TIP packets to trace exceptions, interrupts and other events.

Using the packet trace captured by Intel PT along with the corresponding target program in the binary form, a software developer or a security analyst could fully and perfectly reconstruct the instruction trace pertaining to the execution of the target program. To demonstrate this, we depict the packet trace as well as the target program in disassembly side by side in Figure 4.2. As we can observe from the figure, Intel PT records the address of the entry point with TIP packet TIP 0x400629 and then the conditional jump with a TNT packet indicated by TNT1. Following these two packets, Intel PT also encloses packets TIP 0x05e4 and TIP 0x4006b8 in the packet trace. Using the first two packets shown in the trace, we can easily infer that the program enters its execution at the site 0x400629 and then takes the true branch redirecting the execution from the site 0x40067d to the site 0x400692. As is indicated by consecutive packets TIP 0x05e4 and TIP 0x4006b8, we can further conclude that
the target program invokes a subroutine located at the site 0x05e4 and then returns to the site 0x4006b8

4.3.2 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.

4.3.3 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 [17, 18].

4.3.4 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 [23]. The captured information is organized 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 4.5.

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 4.1. 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 4.1, 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 [133]. 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 4.1. Now, reverse execution can create two hypotheses, one assuming \[ \text{eax} \] and \[ \text{ebp+0x8} \] 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 \( \text{eax} = \text{ebp + 0x8}, \) \( \text{eax} = [\text{ebp + 0x8} + 0x4 \) and \( [\text{eax}] = 0x2. \) For the second hypothesis, the constraint set would include \( \text{eax} \neq \text{ebp + 0x8}, \) \( \text{eax} = [\text{ebp + 0x8} + 0x4 \) and \( [\text{eax}] = 0x2. \) 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 \( \text{eax} = \text{ebp + 0x8} \) 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 4.1 illustrates part of these memory footprints.

With memory footprints recovered, 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, \textit{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 4.1. As is described earlier, in this case, the bad value in register \text{eax} was passed through instruction A19 which copies the bad value from memory \[ \text{ebp-0xC} \] to register \text{eax}. By examining the memory footprints restored, \textit{POMP} can easily find out that the memory indicated by \[ \text{ebp-0xC} \] shares the same address with that indicated by \[ \text{eax} \] in instruction A14. This implies that the bad value is actually propagated from instruction A14. As such, \textit{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.
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.

4.4.1 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.

4.4.1.1 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 4.1, POMP extracts the use and definition from instruction A18 and links them to the head of the chain (see Figure 4.3).

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 4.3).

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, 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 4.3. 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, 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: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 (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 4.1 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 4.4, 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.

As we can observe from the example shown in Figure 4.4, 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 relations. Other techniques, such as static program slicing [134], 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.

### 4.4.1.2 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 $eax \neq ebp+0x8$ and releases the block shown in Figure 4.4, 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 4.3). 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 4.5. After the completion
Figure 4.5: 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 “X” represents the block of a data flow.

of the first step, we assume that our algorithm conservatively treats $A2 \text{ def: } [R_2]$ and $A4 \text{ def: } [R_5]$ 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 recursion depth of two. As we will show in Section 4.6, this setting allows us to perform reverse execution not only in an efficient but also relatively effective manner.

4.4.1.3 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 4.6, 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.

### 4.4.2 Backward Taint Analysis

Recall that the goal of this work is to pinpoint instructions truly pertaining to a program
crash. In Section 4.3, we briefly introduce how backward taint analysis plays the role in
achieving this goal. Here, we describe more details.

To perform backward taint analysis, \textsc{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, \textsc{POMP} deems the program counter (\texttt{eip}) as a
sink because executing an invalid instruction indicates \texttt{eip} carries a bad value. For the
second situation, \textsc{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 4.1. \textsc{POMP} treats register
\texttt{eax} as a sink in that the program crash results from retrieving an invalid instruction from
the address held by register \texttt{eax}.

With a sink identified, \textsc{POMP} taints the sink and performs taint propagation backward.
In the procedure of this backward propagation, \textsc{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 \texttt{eax} serving as the initial
taint variable, \textsc{POMP} selects A19 \texttt{def:eax=[ebp-0xc]} on the chain because this
definition can reach taint variable \texttt{eax} without intervention.

From the definition identified, \textsc{POMP} parses that definition and passes the taint to

\(^3\)Note that an intervening tag placed by a system call blocks only definitions and uses in which a
variable represents a memory access (\textit{e.g.}, \texttt{def:[eax]} or \texttt{use:[ebp]}).
new variables. Since any variables enclosed in a definition could potentially cause
the corruption of the taint variable, the variables which \texttt{POMP} selects and passes the
taint to include all operands, base and index registers (if available). For example, by

\textbf{parsing definition A19:} def:eax=[ebp-0xc], \texttt{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 \texttt{POMP} does not miss the root cause of a program

\textbf{crash though it over-taints some variables that do not actually contribute to the crash. In}

\textbf{Section 4.6, we evaluate and discuss the effect of the over-tainting.}

When passing a taint to a variable indicated by a memory access (\textit{e.g.}, \([R_0]\)), it
should be noted that \texttt{POMP} may not be able to identify the address corresponding to the
memory (\textit{e.g.}, unknown \(R_0\) for variable \([R_0]\)). Once this situation appears, therefore,
\texttt{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 \texttt{def: [R_i]} (where \(R_i\) is a
register).

Similar to the situation seen in reverse execution, when performing taint propagation
backward, \texttt{POMP} may encounter a definition on the chain which intervenes the propa-
gation. For example, given a taint variable \([R_0]\) and a definition \texttt{def: [R_1]} with \(R_1\)
unknown, \texttt{POMP} cannot determine whether \(R_0\) and \(R_1\) share the same value and \texttt{POMP}
should pass the taint to variable \([R_1]\). When this situation appears, \texttt{POMP} follow 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 \texttt{POMP} can pass that taint accordingly. However, in practice, the hypothesis
test may fail to reject. When “fail-to-reject” occurs, therefore, \texttt{POMP} over-taints the
variable in that intervening definition. Again, this can ensure that \texttt{POMP} does not miss
the enclosure of root cause.
4.5 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.

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[112] and libdisasm[114], 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 [135, 136].

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.

### 4.6 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.
### Table 4.2: 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.

#### 4.6.1 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 [108]. Table 4.2 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.

### 4.6.2 Experimental Design

For each program crash shown in Table 4.2, 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 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

76
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.

4.6.3 Experimental Results

We show our experimental results in Table 4.2. Except for test cases Overkill 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 4.4, 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 4.2, 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 4.2, 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 Overkill results from an insufficient PT log. As is specified in Table 4.2, 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.
<table>
<thead>
<tr>
<th>Program</th>
<th>Comparison</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CREDAL</td>
<td>POMP</td>
</tr>
<tr>
<td>Name</td>
<td>LOC</td>
<td># of Inst</td>
</tr>
<tr>
<td>CREDAL</td>
<td>POMP</td>
<td></td>
</tr>
<tr>
<td>coreutils-8.4</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>mcrypt-2.5.8</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>BinUtils-2.15</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>unrtf-0.19.3</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>putils-p17</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>stftp-1.1.0</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>nasm-0.98.38</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>libpng-1.2.5</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>putty-0.66</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Unalz-0.52</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LaTeXX2rtf-1.19</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>aireplay-ng-1.2b3</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>corehttp-0.5.3a</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>gas-2.12</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>abc2mtex-1.6.1</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LibSmi-0.4.8</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>gif2png-2.5.2</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>O3read-0.03</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>unrar-3.9.3</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>nullhttp-0.5.0</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>netutils-1.8</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>nginx-1.4.0</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Python-2.2</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>0verkill-0.16</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Python-2.7</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Average</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Table 4.3: Diagnosis performance comparison between CREDAL and POMP

4.6.4 Comparing POMP with CREDAL

We further perform a study on comparing the performance of CREDAL and POMP, mainly from the perspective of utility and efficiency. In this study, we select the set of cases that are analyzed by both CREDAL and POMP and we present the comparison results in Table 4.3.

Generally speaking, POMP produces better utilities than CREDAL. Among the 25 cases, POMP identifies the root causes for 23 of them while CREDAL can only cover 15. In addition to that, POMP averagely only includes slightly more than 20 instructions while CREDAL on average encloses a segment with more than 19 source code statements (which usually correspond to hundreds of instructions especially when we expand the involved function calls). Presumably, the major reason behind the better utility of POMP
is the feasibility to accurately reconstruct the execution trace.

But zooming into the results, our study in turn reveals that CREDAL has advantages over POMP in other aspects. In the case of aireplay-ng-1.2b3, CREDAL correctly locates the root cause that POMP is unable to capture. The reason behind is that POMP may run into problems when handling memories that are manipulated by system calls, while CREDAL has no such limitations.

More importantly, CREDAL typically only spends seconds on the analysis while POMP may need minutes or even hours to accomplish the diagnosis. This is mainly because of their different analyzing strategies. CREDAL only conducts static analysis which can usually be completed efficiently, while POMP involves recursive hypothesis testing that is very time-consuming.

4.7 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 [20] 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).

### 4.8 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 recon-
struct 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.
Chapter 5  
POMP-V: Improving Postmortem Program Analysis with Static Analysis

5.1 Introduction

Recently, the availability of hardware tracing — Intel Processing Tracing — Significantly facilitates software crash diagnosis, in the sense of enabling reconstruction of the control flow without ambiguity. However, as I have illustrated in the work of POMP [22], successful diagnosis still necessarily needs to understand the data flow during the crashing run. The biggest challenge in reconstructing the data flow comes from uncertain memory aliases. To handle this challenge, POMP proposes the idea of hypothesis testing. Technically speaking, it first makes an assumption about two memory cells (i.e., alias or non-alias) and then explores if this assumption leads to conflict. The arising of conflict indicates that the assumption is incorrect and the opposite is the ground truth.

Hypothesis testing resolves memory alias without errors and POMP demonstrates its utility in handling thousands of instructions. However, it may encounter reduced utility when dealing with long instruction traces. The reason behind is that the information useful for verifying the hypotheses reduces as the trace increases. Gradually, hypothesis
testing becomes insufficient to resolve alias relations. Such a limitation may prevent POMP from producing informative results in more complicated cases. In addition to that, POMP is designed to perform recursive hypothesis testing. This incurs exponentially increasing cost. For instance, the diagnosis in the case of unrar takes around 6 hours. The time consumed is largely beyond the industrial requirement that a case should be processed in a time slot up to seconds.

Over the past decades, many works have proposed to perform alias analysis at the binary code level (e.g., [29–31]). Among all these efforts, value set analysis (VSA) is the most effective and efficient technique [32, 33]. At a high level, VSA subdivides the memory regions into variable-like entities, based on how memory is accessed, and derives an over-approximation of the set of addresses on which variables span. With that, VSA allows two memory references at a program state to be interpreted with respect to the set of possible values that can arise at that state, and thus determines whether they are an alias pair pointing to the same memory cell.

In this dissertation, I enhance POMP with VSA analysis. Technically speaking, we re-use the design of POMP to reconstruct the instruction trace. Against this trace, we perform VSA to obtain the value set pertaining to each memory access. Then I proceed to the reverse execution borrowed from POMP, but we take a different strategy to resolve memory alias — When seeing an uncertain alias, we first query the VSA results for answers. If the uncertainty persists, we switch to hypothesis testing.

The rationale behind the above hybrid design is two-fold. First, VSA purely relies on static information in the binary and such information rarely reduces as the trace becomes longer. As a result, it provides almost constant utilities, regardless of the trace length. This helps resolve a substantial amount of alias relations that hypothesis testing is unable to handle. In turn, resolution of those aliases compensates the information loss during the reverse execution to enable further progress. As we will demonstrate in Chapter 5, this hybrid design significantly improves the reconstruction of data flow. Second, in addition to increasing the utilities, VSA also improves the efficiency of our diagnosis. Alias query
based on the VSA results incurs cost that is negligible in comparison with hypothesis testing. By replacing a large amount of hypothesis testing with VSA queries, we can substantially reduce the diagnosis time.

While the above design is intuitive and straightforward, it is challenging to achieve the full expressiveness of VSA. On the one hand, existing works on VSA mainly develop the theory. This theory, when being strictly followed, always leads to sound results. However, it often produces low precision in practice, mainly because the theory would frequently lose region information to maintain its conservativeness. As a consequence, a substantial number of memory cells have unbounded value sets. To address this challenge, we enhance VSA to infer the region information. Technically, we infer the regions based on well known heuristics. For instance, memory cells using the stack pointer (e.g., esp on x86 platforms) as the base address usually access the stack region. More details are explained in Section 5.3.

On the other hand, VSA is designed to analyze a whole binary while we focus on a single instruction trace (with knowledge about the final execution state). There are two indications behind this difference. First, the instruction trace might be truncated from the middle of a function. This makes the context (e.g., how parameters are prepared) of the beginning function unknown. To address this problem, we augment the truncated first function to be complete and extend our analysis to cover the entire function. Note that we maintain soundness of this analysis. Second, we have more dynamic information, in particular those recovered by our reverse execution. Intuition suggests that such information would escalate the precision of VSA. To this end, we design a recursive algorithm which combines VSA and reverse execution. To be more specific, we perform VSA and reverse execution repetitively until we reach a fixed point of data flow recovery.

In summary, this work makes the following contributions.

• We explore to augment POMP with static binary analysis (in particular value set analysis). Towards this goal, we design an iterative algorithm which combines the static analysis and hypothesis testing to maximize the utility of our reverse
 execution.

- We prototype our design with $\text{POMP-V}$ on Linux to improve both the utility and efficiency of $\text{POMP}$.

- We study a set of 15 cases and compare the results produced by $\text{POMP}$ and $\text{POMP-V}$. It demonstrates that $\text{POMP-V}$ effectively increases memory resolution with the same efficiency or maintains the same level of memory resolution but brings 30x of efficiency improvement.

The rest of this chapter is organized as follows. Section 5.2 describes the necessary background to understand our design. Section 5.3 and Section 5.4 present the design and implementation of $\text{POMP-V}$ in detail. Section 5.5 demonstrates the utility of $\text{POMP-V}$, followed by some discussion on $\text{POMP-V}$ in Section 5.6. Finally, we conclude this work in Section 5.7.
5.2 Background of Value-set Analysis

This work aims at enhancing POMP with VSA to increase its utility and efficiency. In this section, we describe the background of VSA. More specifically, we first introduce the theory of VSA and then give an example to demonstrate how it works.

Value-set analysis is an algorithm designed for analyzing assembly code in a static fashion. Based on the observation that memory layout generally follows, VSA partitions memory into different regions and assigns instructions to the regions, accordingly. For some instructions, VSA achieves region assignment by examining the semantics of the instructions. For example, from a binary code perspective, accesses to global and stack variables appear as \([\text{absolute-address}]\) and \([\text{esp-offset}]\). Thus VSA can easily link the global and stack regions to the instructions \(\text{mov edx, [0x8050684]}\) and \(\text{lea eax, [esp+4]}\), respectively. For other instructions, VSA performs forward data flow analysis to determine the regions tied to instructions in a conservative fashion.\(^1\)

Take for example the assembly code shown in Figure 1a. The instruction in line 7 indicates a write to the target memory \([\text{eax}]\). Through a forward data flow analysis, VSA could easily pinpoint that the value of \(\text{eax}\) is passed through line 2. Given that the instruction \(\text{lea eax, [esp+4]}\) adds the stack pointer by 4 and assigns it to \(\text{eax}\), VSA assigns the stack region to the instruction in line 7.

\(^1\)By “conservative fashion”, we refer to the fact that VSA does not actively infer the value held in a memory cell if the data flow propagation is blocked by an unknown memory reference.
In addition to assigning instructions to memory regions, 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. Take the assembly code shown in Figure 1a as an example which does not involve heap operations. The register *a-locs* contain all the registers, i.e., esp, eax, ebx, ecx, and edx. The global *a-locs* are [0xC4] and [0xC8]. The stack *a-locs* include [esp], [eax] and [ebx].

It should be noticed that, as is illustrated in Table 5.3, VSA represents a stack *a-loc* as a combination of the value held by a memory cell and the value set indicating the address of that memory cell. For example, the instruction `mov [esp], eax` accesses the stack memory, and VSA specifies its corresponding stack *a-loc* as [esp] (⊥, [-44, -44], ⊥). Here, [esp] indicates the name of the stack memory cell, and (⊥, [-44, -44], ⊥) is the value set of the memory address. Or in other words, the values that register esp could potentially equal to at the site of that instruction.

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 3-tuple pertaining to the three regions partitioned. For each element in the tuple, VSA specifies a range of offsets which indicates the values that the *a-loc* could 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 5.3, VSA specifies its value set as a 3-tuple (global → ⊥, stack → [-44, -44], heap → ⊥), for brevity (⊥, [-44, -44], ⊥). In this set, ⊥ is a symbol – denoting the empty set of offsets (i.e. ∅) – reflecting the fact that the register esp is the stack pointer in x86 architecture and cannot refer to any memory cells on the heap or global region. Since the semantics of the first instruction is to offset esp by 44 from the starting point of the stack, VSA assigns the value set {−44} to the register *a-loc* esp, and attaches this set to the stack. It should be noticed that for specification consistency

---

2Except for registers, VSA considers only memory dereferences as *a-locs*. In x86 architecture, the `lea` instruction does not involve memory dereference, and therefore [esp+4] and [esp+24] are not treated as stack *a-locs*. 

89
<table>
<thead>
<tr>
<th></th>
<th>Complete Trace</th>
<th>Incomplete Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>\textit{A-loc}</td>
<td>Value-set</td>
</tr>
<tr>
<td>11</td>
<td>\textit{esp}</td>
<td>(⊥, [-44, -44], ⊥)</td>
</tr>
<tr>
<td>22</td>
<td>\textit{eax}</td>
<td>(⊥, [-40, -40], ⊥)</td>
</tr>
<tr>
<td>33</td>
<td>\textit{ebx}</td>
<td>(⊥, [-20, -20], ⊥)</td>
</tr>
<tr>
<td>44</td>
<td>\text{[esp]}</td>
<td>(⊥, [-44, -44], ⊥)</td>
</tr>
<tr>
<td>55</td>
<td>\textit{ecx}</td>
<td>([0, 0], ⊥, ⊥)</td>
</tr>
<tr>
<td>L1L1</td>
<td>[0xC4]</td>
<td>([0, 0], ⊥, ⊥)</td>
</tr>
<tr>
<td></td>
<td>([0xC4, 0xC4], ⊥, ⊥)</td>
<td>([0, 0], ⊥, ⊥)</td>
</tr>
<tr>
<td>L1L2</td>
<td>\textit{edx}</td>
<td>([0, 0], ⊥, ⊥)</td>
</tr>
<tr>
<td></td>
<td>([0, 0], ⊥, ⊥)</td>
<td>([0, 0], ⊥, ⊥)</td>
</tr>
<tr>
<td>77</td>
<td>\text{[eax]}</td>
<td>([0, 0], ⊥, ⊥)</td>
</tr>
<tr>
<td></td>
<td>(⊥, [-40, -40], ⊥)</td>
<td>([0, 0], ⊥, ⊥)</td>
</tr>
<tr>
<td>88</td>
<td>[0xC8]</td>
<td>([0, 0], ⊥, ⊥)</td>
</tr>
<tr>
<td></td>
<td>([0xC8, 0xC8], ⊥, ⊥)</td>
<td>([0, 0], ⊥, ⊥)</td>
</tr>
<tr>
<td>L1L3</td>
<td>\textit{edx}</td>
<td>([0, 0], ⊥, ⊥)</td>
</tr>
<tr>
<td></td>
<td>([0, 0], ⊥, ⊥)</td>
<td>([0, 0], ⊥, ⊥)</td>
</tr>
<tr>
<td>99</td>
<td>\text{[ebx]}</td>
<td>([0, 0], ⊥, ⊥)</td>
</tr>
<tr>
<td></td>
<td>(⊥, [-20, -20], ⊥)</td>
<td>([0, 0], ⊥, ⊥)</td>
</tr>
<tr>
<td>1010</td>
<td>\textit{eax}</td>
<td>([0, 0], ⊥, ⊥)</td>
</tr>
<tr>
<td>1111</td>
<td>\textit{ebx}</td>
<td>([0, 0], ⊥, ⊥)</td>
</tr>
<tr>
<td>1212</td>
<td>\textit{ecx}</td>
<td>([0, 0], ⊥, ⊥)</td>
</tr>
<tr>
<td>1313</td>
<td>\text{-}</td>
<td>\text{-}</td>
</tr>
<tr>
<td>1414</td>
<td>\text{-}</td>
<td>\text{-}</td>
</tr>
<tr>
<td>1515</td>
<td>\text{[esp+4]}</td>
<td>([0, 0], ⊥, ⊥)</td>
</tr>
<tr>
<td></td>
<td>(⊥, [-40, -40], ⊥)</td>
<td>([0, 0], ⊥, ⊥)</td>
</tr>
</tbody>
</table>

\textbf{Figure 5.3:} \textit{A-locs} and value-sets corresponding to complete and incomplete traces.

we write the value sets \{−44\} tied to the stack as \([-44, -44]\).

\subsection*{5.3 Design}

In this section, we present the design of \textit{POMP-V}. We first explain our approach to infer the regions of memory accesses and then describe the details of our VSA analysis. Finally, we present the algorithm to combine VSA with reverse execution.
mov ebp, esp ;set up frame pointer
... ...
mov edx, [ebp-0x24] ; access stack
... ...
lea edx, [ebp-0x10]
... ...
mov [edx], 0x00 ; access stack
... ...

(a) Reference to stack

... ...
ret ;return from malloc
lea edi, [eax] ;receives buffer address
mov [edi+0x4], eax ;access heap
... ...

(b) Reference to heap

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

(c) Reference to global variables

Figure 5.4: Examples of memory references to different regions
5.3.1 Identifying Memory Regions

Generally speaking, compilers often generate code that uses different patterns to access different memory regions. This gives us the foundation to do region inference — We seek for those patterns contained in the instruction trace and map them into region information. In the following, we summarize the patterns we use in our design.

5.3.1.1 Stack

A function usually maintains a frame pointer in a register (e.g., ebp on x86-32) and uses the frame pointer as the base_reg for accesses to stack. Figure 5.4a shows a very common example of such use. Instruction 1 preserves the stack frame in ebp and Instruction 3 uses ebp as the base_reg for an access to stack. The frame pointer could also propagate with the execution and later be used to access variables on stack. In Figure 5.4a, Instruction 5 stores the frame pointer plus an offset into edx. Going with the execution, edx reaches Instruction 7 and is used as base_reg for another stack access. We keep track propagation of the frame pointer when possible so that we can identify further stack accesses. In functions that do not use a frame pointer, typically the stack pointer (e.g., esp on x86-32) plays the role of a frame pointer, with which we can also identify stack accesses by following the above process.

5.3.1.2 Heap

In practice, programs typically rely on standard heap allocators (e.g., ptmalloc) to manage memory on heap. They use functions such as malloc and calloc to allocate buffers on heap and get the starting addresses of new buffers from their return values. To access contents in a heap buffer, a program obtains the base_reg by adding an offset to the starting address of the buffer. Figure 5.4b shows an access to heap. Instruction 2 returns from malloc with a new buffer allocated on the heap and Instruction 3 stores the return value of malloc (i.e., the starting address of the new buffer) into edi. Instruction
accesses this heap buffer by taking edi plus offset 0x4 as base_reg. Similar to the frame pointer, the return value of an allocation function may also propagate with the execution and be used as base_reg later. Likewise we will also keep track of such propagation when possible.

5.3.1.3 Global variables

On a Linux system, accesses to global variables depend on the type of the program. In programs that are not compiled as Position Independent Code (PIC), addresses of global variables are encoded in instructions. The region being accessed is explicitly presented. In a program compiled as PIC, two redirections are performed to obtain the address of a global variable. We use the example in Figure 5.4c to briefly explain the process. At first, the value of the current Program Counter (PC) is retrieved by a routine function (Instruction 2 calls get_pc_thunk.cc to store PC into ecx). Then PC is added with a constant value to obtain the base address of the Global Offset Table (GOT) (Instruction 3 saves into ecx the base address of GOT). Finally, the GOT entry that stores the address of the global variable is visited and the address is retrieved (Instruction 4 visits the entry in GOT with an offset 0x10 and gets the address stored inside; Instruction 5 finally accesses the global variables). In particular, whenever a global variable is accessed, the above process will be repeated and thus, the base_ptr of a global variable is usually not propagated.

5.3.1.4 Static variables and TLS variables

The accesses to static variables and TLS variables are similar to those to global variables. In PIC code, accesses to static variables also rely on the PC but have no reference to GOT. While for an access to a TLS variable, obtaining its address not only needs the PC and the GOT, but also requires the value in a thread-specific register (e.g., gs in x86-32). Due to limited space, we will skip the details, but it is worthy mentioning that the mechanisms of accessing static variables and TLS variables are unique enough to be differentiated.
Besides the above rule, we additionally apply another strategy to determine the regions of memory accesses [137]. The insight is that many standard library functions only write to specific memory regions. We identify three categories of those functions in the C standard library libc.so and the Linux dynamic linker ld-linux.so. The first category of functions never write to memory or only modify contents on their stack frames, such as getc and getpid. The second category of functions, besides writing to the associated stack frames, may also modify a special TLS variable errno (e.g., lseek and listen). The last category of functions not only write to the areas aforementioned, but also modify some special regions that other functions will never touch. Particular examples are libc functions that serve heap allocations and ld-linux functions that perform dynamic linking. Heap allocation functions only modify meta-data for heap management and dynamic linking functions only access certain sections in the data segment (e.g., .got and .hash). For a memory access (especially memory write) in one of those functions, we can easily determine the region(s) that are potentially targeted.

### 5.3.2 Value Set Analysis

Given an instruction trace and the region information we identify, VSA can track down a-locs, derive value sets, and perform memory alias analysis by examining the value set tied to each of the a-locs. To illustrate this, we take the assembly code depicted in Figure 1a as an example and assume they represent the complete execution trace of a program. Supposing that Figure 5.3 indicates the value set tied to each of the a-locs identified from the assembly code, we can easily observe that memory [eax] and [esp+4] are the only pair of memory alias 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. (⊥, [-40, -40], ⊥). To better understand the effect of VSA on alias analysis, we derive all the alias and non-alias relationships from the value sets specified in Table 5.3, and depict them in the upper triangular portion of the matrix shown in Table 5.2.
In the example above, VSA exhibits perfect performance in alias analysis. However, this does not imply that VSA could perfectly – or significantly – resolve the memory alias issue in the context of postmortem program analysis. To demonstrate this, we again take as example the assembly code shown in Figure 5.1. However, different from the setup specified above, we assume the trace is truncated and available only starting from line 6 for the following reason. Execution tracing is typically limited by the available storage. By truncating the trace, we can emulate a scenario commonly found during failure diagnosis. That is, the execution trace logged for failure diagnosis has a limited length, indicating only a partial execution chronology prior to a program crash.

In Figure 5.3, we also show all the a-locs identified from this truncated trace. Compared with the value set derived from the full execution trace shown in the same table, we can easily observe that all the value sets tied to the a-locs are varied. This is because VSA performs an over-approximation in value-set construction and the missing context limits the capability of VSA with respect to reasoning memory regions or offsets within a region. Take the a-loc indicated by [eax] (T, T, T) as an example. Without the execution context, VSA conservatively assumes eax could equal to any value. Thus, memory [eax] could refer to any memory regions with an arbitrary offset indicated by the symbol T. Through forward data flow analysis, VSA could find that the value of [eax] is passed through the instruction in line 6, which indicates the operation of retrieving a value from a global region. Since the incomplete trace does not provide VSA with sufficient context to infer the value held in [0xC4], the value set tied to this a-loc can be represented as (T, T, T). From the a-locs identified from the truncated trace along with their value set, we follow the aforementioned approach to examine value set intersection, and illustrate the alias and non-alias relationships in the lower triangular portion of the matrix shown in Figure 5.2. As we can easily observe, without the full execution trace, VSA over-approximates value sets tied to a-locs, and conservatively deems 60% of memory pairs as may-alias relationships. Since may-alias represents uncertainty relationship, Figure 5.2 illustrates them as the question symbol ‘?’.
Algorithm 1 Recursive VSA enhanced reverse execution algorithm. Each round of reverse execution only performs one layer of hypothesis testing.

**INPUT:**
- `inst_trace` - The instruction trace
- `core_dump` - The core dump

**OUTPUT:**
- Data flow along the instruction trace `data_flow`

```plaintext
1: procedure UPDATE_TRACEBITS
2:     data_flow = 0
3:     temp_df = 0
4:     temp_df = rev_exe_nohypo(temp_df)
5:     while temp_df > data_flow do
6:         data_flow = temp_df
7:         temp_df = do_vsa(temp_df)
8:         temp_df = do_reverse_exe(temp_df)  ▶ A round of reverse execution without hypothesis testing.
9:     end while
10:    data_flow = temp_df
11: end procedure
```

such results to derive the data flow for failure diagnosis renders the diagnosis incapable of yielding any useful results, for the simple reason that the data flow can potentially be highly imprecise.

To address the above problem, we enlarge the instruction trace to cover 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 5.1, there is a single path from the entry to the trace beginning. Therefore, we can easily reconstruct the full path (as shown in the upper half) 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.

**5.3.3 Combining VSA and Reverse Execution**

Our goal of using VSA is to facilitate the reverse execution developed in POMP. To achieve this goal, we design a recursive algorithm as shown in Algorithm 1. Given an instruction trace and the corresponding core dump, we initially do a round of reverse execution without hypothesis testing. This helps initialize a preliminary data flow. Then
we enter a recursive analysis loop. During each iteration of this loop, we first perform a round of VSA and then conduct another round of reverse execution. When encountering uncertain alias relations during this reverse execution, we first query the VSA results and then switch to hypothesis testing if necessary. When an iteration cannot increase the data flow, which means the combination of VSA and reverse execution reaches a fixed point, we terminate the loop and report the final results. This algorithm pursues the maximal utility, since it ensures both VSA and hypothesis testing reach their maximal expressiveness.

5.3.4 Further Improvements

Going beyond the VSA analysis, we further identified two other improvements via static analysis. First, we observe that the frame pointer inside a function — if this function maintains a frame pointer — keeps a constant value. This enables us to propagate the frame pointer across the whole function. Second, certain memory regions are mapped as non-writable. Such regions can never alias with those writable regions.

5.3.4.1 Constancy of frame pointers

As is aforementioned, many functions maintain a frame pointer for the convenience of stack access. Inside the function body (i.e., the code from the end of prologue to the beginning of epilogue), the frame pointer holds a fixed value. Even when the execution crosses a child function, the calling convention will guarantee the constancy of the frame pointer. Based on this, we add a constraint that all uses of the frame pointer register in the body of a function share the same value. When one use of the frame pointer is resolved, we can quickly assign the value to the remaining; When the alias between one use of the frame pointer and a memory access is known, we can likewise quickly propagate the alias information. As a large portion of memory accesses are to stack, the benefit of these two kinds of quick propagation is significant.
5.3.4.2 Non-writable memory.

For security reasons, certain memory regions are set as non-writable. On Linux systems that use Executable and Linkable Format (ELF), the code segment (`.text`) are non-writable to achieve Data Execution Prevention. An ELF file typically also includes a `.rodata` section to store read-only objects (e.g., format strings and jump tables). Due to the memory-access restriction, an access to a non-writable memory region can never alias with any memory write.

5.4 Implementation

We have implemented a prototype of POMP-V 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 is built on the top of POMP. We customized POMP to plug in a VSA module which we developed to perform analysis against an execution trace indicating a single control flow. To be more specific, we implemented an instruction parser using `libdisasm` and 84 distinct instruction handlers to perform value-set calculation. In total, our VSA implementation contains about 9,500 lines of C code. It should be noticed that the value-set calculation for instructions with similar semantics (e.g., `ja`, `jb`, `jc`) were taken care of by a unique handler.

Recall that our 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 [138].

98
<table>
<thead>
<tr>
<th>Program</th>
<th>Size (LoC)</th>
<th>CVE-ID</th>
<th>Type</th>
<th>Trace length</th>
<th>Root cause VSA-POMP</th>
<th>Root cause HT-POMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc2mtex-1.6.1</td>
<td>4052</td>
<td>NA</td>
<td>Stuck overflow</td>
<td>29521</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>aireplay-ng-1.2b3</td>
<td>62656</td>
<td>2014-8322</td>
<td>Stack overflow</td>
<td>18569</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ClamAV-0.93.3</td>
<td>771540</td>
<td>2008-5314</td>
<td>Stack Exhaustion</td>
<td>1000000</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>gas-2.12</td>
<td>595504</td>
<td>2005-4807</td>
<td>Stack overflow</td>
<td>25713</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>gif2png-2.5.2</td>
<td>1331</td>
<td>2009-5018</td>
<td>Stack overflow</td>
<td>70854</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LaTeX2pdf-1.9</td>
<td>14473</td>
<td>2004-2167</td>
<td>Stack overflow</td>
<td>17056</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>libpng-1.2.5</td>
<td>33681</td>
<td>2004-0597</td>
<td>Stack overflow</td>
<td>6026</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>libSMI-0.4.8</td>
<td>80461</td>
<td>2010-2891</td>
<td>Stack overflow</td>
<td>50787</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>nasm-0.98.38</td>
<td>33553</td>
<td>2004-1287</td>
<td>Stack overflow</td>
<td>4064</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>nginx-1.4.0</td>
<td>100255</td>
<td>2013-2028</td>
<td>Integer overflow</td>
<td>158</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>npd-4.2.6</td>
<td>1849</td>
<td>2002-1496</td>
<td>Heap overflow</td>
<td>141</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>nulltmp-0.5.9</td>
<td>1849</td>
<td>2002-1496</td>
<td>Heap overflow</td>
<td>141</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>O3read-0.03</td>
<td>932</td>
<td>2004-1288</td>
<td>Stack overflow</td>
<td>78244</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>coreutils-8.4</td>
<td>138135</td>
<td>2013-0222</td>
<td>Stuck overflow</td>
<td>50</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>prozilla-8.4</td>
<td>138135</td>
<td>2013-0222</td>
<td>Stuck overflow</td>
<td>50</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 5.1: 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-V reversely executed. Root Cause VSA-POMP shows if we can capture the root cause when only using VSA for alias verification while Root Cause HT-POMP shows if we can capture the root cause when only using hypothesis testing for alias verification.

5.5 Evaluation

In this section, we demonstrate the utility of POMP-V using the crashes resulting from real-world vulnerabilities. To be more specific, we present the utility and efficiency of POMP-V, and inspect how different strategies used by POMP-V impact the results.

5.5.1 Evaluation Settings

To demonstrate the utility of POMP-V, we selected 15 programs and benchmarked POMP-V with their crashes resulting from the corresponding real-world PoCs obtained from Offensive Security Exploit Database Archive [108].

Table 5.1 shows these crashing programs and summarizes the corresponding vulnerabilities. Similar to our evaluation on POMP, we select programs that cover a wide spectrum ranging from sophisticated software like gas with lines of code over 590K to lightweight software such as O3read 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 stack exhaustion. The reason behind this selection is to demonstrate that, beyond memory corruption vulnerabilities, POMP-V can be generally applicable to other kinds of software defects. For each of these crashes, we also manually identify the root cause.

By design, POMP-V enhances POMP with static VSA to increase its utility in rebuilding data flow and reduce the computation complexity. Therefore, our evaluation mainly focuses on these two most important metrics. In the following, we present the evaluation details.

5.5.2 Utility Evaluation

The core idea of POMP-V is introducing VSA into postmortem program analysis. The most intuitive utility indicator is its effectiveness in tracking down the root cause. To serve this evaluation, we perform two sets of experiments, one performing reverse execution with VSA only and one performing reverse execution with hypothesis testing only. Following that, we leverage the backwards tainting analysis to locate the root cause. As demonstrated in Table 5.1, VSA and hypothesis testing exhibit identical utility in capturing the root causes. They both succeeded in 14 of the cases and only missed the case of aireplay-ng. As we have previously explained, in the test case of aireplay-ng, 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 cannot restore, the reverse execution treats sys_read as a “super” intervening tag that blocks the propagation of many definitions, making the output of POMP and POMP-V less informative for the diagnosis.

The above evaluation shows that VSA has similar capability to hypothesis testing, in terms of enabling reverse execution to track down the root causes. However, this does mean they internally have identical utilities. To take a closer look at their difference, we perform another measurement, in which we quantify the data flow VSA and hypothesis testing enable reverse execution to rebuild. To be more specific, when the VSA enabled
reverse execution and hypothesis testing enabled reverse execution terminate, 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 extend that we understand use-define relations ³.

In this measurement, we include not only the VSA and hypothesis testing enabled reverse execution, but also the reverse execution with one round of VSA plus hypothesis testing and the reverse execution following Algorithm 1. The measurement results are shown in Figure 5.5.

As is shown in Figure 5.5, VSA outperforms hypothesis testing in several cases, in particular in nasm and prozilla. Presumably, this is mainly due to that the dynamic information left in the core dump is insufficient to verify aliases that can be determined by the static analysis. However, hypothesis testing performs better in the case of libpng and libsmi. This in turn illustrates that dynamic information can be more beneficial in certain cases. In conclusion, this demonstrates an intuition that VSA and hypothesis testing have distinct characteristics and advantages. To verify this intuition, we further run reverse execution with both VSA and hypothesis testing. To be more specific, when we encounter uncertain alias relations, we first query VSA and then do hypothesis testing if necessary. As illustrated in Figure 5.5, this hybrid approach outperforms both VSA and hypothesis testing in all the cases. This well supports the above intuition. Going beyond this, we design a recursive algorithm as shown in Algorithm 1, which runs multiple rounds of the combination until a fixed point is reached. Figure 5.5 also presents the results of using this recursive algorithm. For all the cases, we reached the fixed point within two rounds and achieved the highest utilities. Summarizing the above measurement, VSA — in terms of facilitating alias verification in general — has similar expressiveness to hypothesis testing. However, VSA and hypothesis also have unique strengthens. A hybrid approach to combine them leads to the optimal utility.

³Use-define relations depend on our knowledge about the addresses of memory accesses.
Figure 5.5: Measurement of recovered memory addresses. HT refers to reverse execution with hypothesis testing. VSA indicates reverse execution with VSA. VSA+HT shows reverse execution with one round of VSA plus HT while Fix Point shows reverse execution that follows Algorithm 1 to reach a fixed point. Ground Truth refers to reverse execution with ground truth of all alias relations.

5.5.3 Efficiency Measurement

Going beyond increasing the utility of reverse execution, we also aim to achieve a better efficiency via introducing VSA. To evaluate the efficiency improvement, we perform another round of evaluation. In this evaluation, we measure the computation complexity of the above experiments and show the results in Table 5.2.

As shown by the evaluation, VSA takes significantly less time than hypothesis testing. On average, the computation complexity of VSA is $30\times$ less than hypothesis testing. This well supports our intuition that static analysis has superior efficiency. Considering that VSA and hypothesis testing provide similar utility, this evaluation also indicates that
<table>
<thead>
<tr>
<th>Name</th>
<th>HT</th>
<th>VSA</th>
<th>VSA + HT</th>
<th>Fix Point</th>
<th>Ground Truth</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc2mtex-1.6.1</td>
<td>120 min</td>
<td>3 min</td>
<td>21 min</td>
<td>38 min</td>
<td>3 min</td>
</tr>
<tr>
<td>aireplay-ng-1.2-beta3</td>
<td>10 min</td>
<td>5 min</td>
<td>8 min</td>
<td>8 min</td>
<td>8 min</td>
</tr>
<tr>
<td>clamav-0.93.3</td>
<td>1149 min</td>
<td>25 min</td>
<td>2010 min</td>
<td>2500 min</td>
<td>16 min</td>
</tr>
<tr>
<td>gas-2.12</td>
<td>46 min</td>
<td>10 min</td>
<td>11 min</td>
<td>13 min</td>
<td>10 min</td>
</tr>
<tr>
<td>gif2png-2.5.2</td>
<td>31 min</td>
<td>8 min</td>
<td>48 min</td>
<td>63 min</td>
<td>8 min</td>
</tr>
<tr>
<td>latex2rft-1.9.15</td>
<td>8 min</td>
<td>7 min</td>
<td>8 min</td>
<td>8 min</td>
<td>8 min</td>
</tr>
<tr>
<td>libpng-1.2.5</td>
<td>6 min</td>
<td>4 min</td>
<td>6 min</td>
<td>7 min</td>
<td>4 min</td>
</tr>
<tr>
<td>libsmi-0.4.8</td>
<td>705 min</td>
<td>12 min</td>
<td>857 min</td>
<td>1052 min</td>
<td>11 min</td>
</tr>
<tr>
<td>nasm-0.98.38</td>
<td>1 min</td>
<td>3 sec</td>
<td>6 sec</td>
<td>6 sec</td>
<td>3 sec</td>
</tr>
<tr>
<td>nginx-1.4.0</td>
<td>3.2 sec</td>
<td>3.3 sec</td>
<td>3.3 sec</td>
<td>3.4 sec</td>
<td>3.2 sec</td>
</tr>
<tr>
<td>ntpd-4.2.6</td>
<td>3.7 sec</td>
<td>0.3 sec</td>
<td>1 min</td>
<td>1 min</td>
<td>0.3 sec</td>
</tr>
<tr>
<td>nullhttpd-0.5.0</td>
<td>0.2 sec</td>
<td>0.1 sec</td>
<td>0.3 sec</td>
<td>0.4 sec</td>
<td>0.1 sec</td>
</tr>
<tr>
<td>o3read-0.03</td>
<td>2332 min</td>
<td>23 min</td>
<td>2973 min</td>
<td>3573 min</td>
<td>22 min</td>
</tr>
<tr>
<td>prozilla-1.3.6</td>
<td>2 min</td>
<td>1 min</td>
<td>2 min</td>
<td>2 min</td>
<td>1 min</td>
</tr>
<tr>
<td>coreutils-8.4</td>
<td>122 min</td>
<td>45 min</td>
<td>175 min</td>
<td>238 min</td>
<td>89 min</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>302 min</td>
<td>10 min</td>
<td>408 min</td>
<td>500 min</td>
<td>12 min</td>
</tr>
</tbody>
</table>

**Table 5.2:** Measurement of reverse execution efficiency. HT refers to reverse execution with hypothesis testing. VSA indicates reverse execution with VSA. VSA+HT shows reverse execution with one round of VSA plus HT while Fix Point shows reverse execution that follows Algorithm 1 to reach a fixed point.

VSA is a better choice in time critical scenarios. Going further, we also measure the time complexity of our combination of VSA and hypothesis testing as well as our fix-point algorithm (Algorithm 1). In the combination based reverse execution, we first query the VSA and then try hypothesis testing when resolving alias relations. By intuition, this shall avoid a great many of hypothesis testing and would reduce the computation complexity. However, our evaluation shows that the combination takes more time than using hypothesis testing only. This does not mean our design insight is incorrect. Instead, we believe this is mainly because VSA recovers more information that can be consumed by hypothesis testing and in turn trigger further space of reverse execution. This is actually supported by the above utility evaluation — This combination leads to better utility than VSA only reverse execution and hypothesis testing only reverse execution. Last, our algorithm to reach the fixed point introduces the highest computation overhead. Again this is understandable as it involves the most computations.
5.6 Discussion and Future Work

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

5.6.1 Memory Corruption

POMP-V introduces VSA into our reverse execution, which brings better utility and higher efficiency. In order to improve the precision of VSA, we rely on a set of patterns to determine region information. However, when memory corruption happens and leads to a software crash, the data integrity of the execution might be corrupted and the patterns we follow may be invalid. For instance, we consider a memory address indexed by esp as an access to the stack. But when memory corruption happens, esp could get contaminated and no longer points to the stack. In this case, our VSA would make a mistake. Therefore, in diagnosing crash due to memory corruption, following our design and using VSA may introduce errors. Although we did not encounter such errors in our evaluation, we believe occurrence of such errors is possible and we suggest avoiding to use VSA when reverse execution is used to deal with a memory corruption induced software crash. In the future, we will explore how to avoid the above mentioned errors when VSA is under deployment.

5.6.2 Multiple Threads

As an extension of POMP, POMP-V also focuses only on analyzing the post-crash artifact produced by a crashing thread. As a consequence, we assume the root cause of the crash is enclosed within the instructions executed by the crashing thread and other threads do not intervene its execution. In many cases, this assumption may not hold, and our current diagnosis approach may not work and even mislead root cause identification. As we discussed in Chapter 4, by simply augmenting our process tracing with the capability of
recording the timing of execution, we can synthesize a complete execution trace, making POMP-V working properly. As part of the future work, we will integrate this extension into the next version of POMP-V.

5.6.3 Further Optimization

As we have demonstrated through our evaluation, the combination of VSA and hypothesis testing could bring the best utility to reverse execution. However, it also brings extra computation complexity. We observe that a major cause of this complexity is that we perform VSA and hypothesis testing independently during each iteration, which involves plenty of repetitive and dummy operations. In the future, we will explore techniques to merge VSA and hypothesis testing, in which we will seek to reduce those unnecessary operations.

5.7 Conclusion

In this work, we introduce VSA into the reverse execution developed in POMP. To achieve higher precision, we enhance VSA with pattern-based reference of region information and the capability of dealing with truncated instruction traces. We show that VSA can provide equal (if not better) utility as hypothesis testing while it has substantially lower computation complexity. By combing VSA and hypothesis testing during reverse execution and iterating them to reach a fixed point, we can achieve the best utility but incurs the greatest slowdown. Following these findings, we conclude that static analysis, in particular VSA, can facilitate reverse execution from the perspective of either utility or efficiency.
Chapter 6  |  Conclusion and Future Work

This dissertation research relies on information left over by a vulnerability-induced software crash and program analysis techniques to automatically locate the vulnerability. In this research, I explore the challenges to achieve automated diagnosis and develop two techniques to overcome these challenges. The first technique combines information in the crashing state and analyses on the source code to locate the code segments that contain the vulnerability, and the second technique leverages a new hardware feature on recent Intel Processors to accurately pinpoint the vulnerabilities.

In Chapter 3, I present the first technique CREDAL, which facilitates core dump analysis. With the support from source code, I show that CREDAL can enhance core dump analysis and make a core dump more informative for diagnosing software defects, particularly locating memory corruption vulnerabilities. The design of CREDAL follows a conservative principle. Thus, it preserves the utility of a core dump, and at the same time, minimizes the uncertainty in core dump analysis. I demonstrate the utility of CREDAL using the crashes corresponding to 73 memory corruption vulnerabilities. I show that CREDAL can accurately pinpoint a crash point as well as a stack trace. In addition, I demonstrate that CREDAL can capture a memory corruption vulnerability, which typically lies in the code fragment relevant to data corruption. Following this finding, I safely conclude CREDAL can significantly downsize the code space that a software developer (or security analyst) needs to manually examine, especially when
memory corruption occurs.

In Chapter 4, I explain POMP, the second technique developed for Linux system to analyze post-crash artifacts. I show that POMP can significantly reduce the manual efforts on the diagnosis of program errors, 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.

In Chapter 5, I enhance POMP with the VSA analysis. Technically speaking, I re-use the design of POMP to reconstruct the instruction trace. Against this trace, I perform VSA to obtain the value set pertaining to each memory access. Then I proceed to the reverse execution borrowed from POMP, but I take a different strategy to resolve memory alias — When seeing an uncertain alias, I first query the VSA results for answers. If the uncertainty persists, I switch to the original hypothesis testing. This work demonstrates that static analysis, in particular VSA, could facilitate reverse execution from the perspectives of both utility and efficiency.

In conclusion, this dissertation research develops a general idea of automating software crash diagnosis — Utilizing all types of information left behind a software crash, I first reconstruct the control flow and data flow prior to the crash and then seek for vulnerabilities by locating security violations. This dissertation research also provides insights to implement this idea by showcasing CREDAL and POMP (and POMP-V) with details of their design and development.

Building on the top of my dissertation research, I would like to further seek in advancement for automated software crash diagnosis. To be more specific, I will explore along two lines and in the following, I introduce them with details.

Both CREDAL and POMP are designed to handle vulnerabilities that pertain to the crashing thread. But in practice, a large number of software crashes are caused by race condition bugs, which are typically triggered by incorrect inter-leavings between
different threads. In the future, I plan to extend \texttt{POMP} to deal with software crashes that are induced by concurrent bugs. My insight is as follows. Intel PT can not only trace the control flow transfers, but also log the time-stamp associated with each transfer. This enables \texttt{POMP} to roughly understand the execution schedules of different threads. By reversely executing each individual thread, ideally, \texttt{POMP} may learn the inter-leavings of memory accesses among different threads and further capture atomicity violations (which are usually the locations of concurrent bugs). To implement the above idea, \texttt{POMP} needs extensions to overcome several major challenges. First, Intel PT can attach at most one time-stamp to each control flow transfer. Between two control flow transfers (i.e., two time-stamps), a large amount of instructions could be executed. Therefore, I can only learn the inter-leavings among different threads at a coarse granularity. We still need extra efforts to pinpoint the exact inter-leaving in a relatively large space. Second, different threads share memory via global variables. To build data flow related to those global variables, we necessarily need to simultaneously consider different threads. This introduces a great challenge to the reverse execution — We have to consider the alias relations between memory accesses incurred in different threads and our hypothesis testing will need to go across the boundary of threads.

\texttt{POMP} relies on Intel PT to trace the instructions and performs reverse execution to recover the data accesses. In practice, this reverse execution may recover insufficient information to identify the root cause or take very long time to finish. The major reason is the lack of information about data accessing. Inspired by this, I plan to explore a better \texttt{POMP} by utilizing Embedded Trace Macro-cell (ETM). ETM is a hardware feature equipped on ARM platforms. It can trace not only the control flow transfers, but also the addresses of memory accesses. Intuitively, the traces from ETM will enable us to quickly reconstruct the full control flow and data flow prior to the software crash. This will guarantee accurate diagnosis. But to adopt this idea in practice, we need to address two critical challenges. On the one hand, ETM is different from Intel PT in the configuration of tracing buffer. ETM provides a small fixed-size physical buffer to store the tracing
data. This buffer can quickly get occupied and when the buffer is full, no interrupt will raise. To maintain a continuous trace, we need a mechanism to timely migrate the tracing data before the buffer gets full. On the other hand, tracing both of control flow and data flow will result in quicker consumption of tracing storage and incurs more frequent trace migration. This leads to another problem. That is, we need to achieve a good trade-off between comprehensiveness of tracing and storage overhead.
Bibliography


[109] “Analysis of nginx 1.3.9/1.4.0 stack buffer overflow and x64 exploitation (CVE-2013-2028),” http://www.vnsecurity.net/research/2013/05/21/analysis-of-nginx-cve-2013-2028.html.


Jun Xu enrolled in the Ph.D. program in Information Sciences and Technology at Pennsylvania State University in August 2013. Prior to that, he received his B.S. degree in Information Science and Technology from University of Science and Technology of China in June 2013. His research interests include system security and software security. His work has led to 9 peer-reviewed papers published in conferences and journals including USENIX Security, ACM CCS, ACM SenSys, DSN, ESORICS, etc. He is a recipient of the 2018 Penn State Alumni Association Dissertation Award and the 2017 RSA Conference Security Scholarship.