Protection from Within: Runtime Hardening Techniques for COTS Binaries

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Mohamed Elsabagh
Master of Science
Nile University, Egypt, 2011
Bachelor of Science
Alexandria University, Egypt, 2009

Director: Dr. Angelos Stavrou, Professor
Department of Computer Science

Summer 2017
George Mason University
Fairfax, VA
Copyright © 2017 Mohamed Elsabagh

This material is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/

All trademarks, software, services, and vendors mentioned in this work are the property of their respective owners.
Dedication

إلى من مهر الطريق بعز الله...
إلى من زللت المعاب برعايتها...
إلى من رافقتني الطريق...
إلى من موايد العون...
Acknowledgments

I would like to thank Drs. Angelos Stavrou, Daniel Barbará, Duminda Wijesekera, Hakan Aydin, and Houman Homayoun for serving on my committee and for their feedback.

I am especially indebted to Dr. Angelos Stavrou whose guidance, expertise, and endless support shaped my experience as a doctoral student. Dr. Stavrou worked actively to provide me with all the support, time, funding, and flexibility needed to pursue my goals.

I am grateful to all whom I have had the pleasure of working with during my doctorate. I would especially like to thank Dr. Daniel Fleck for leading by example and showing great character. My gratitude extends to Dr. Daniel Barbará for his contributions and valuable advice, and for taking the time to aid me with some of the machine learning constructs used in this work. I would also like to thank Michael Kaplan and Thomas Bowen from Vencore Labs for their generous sharing of benchmarks and experimental results.

I want to thank Patricia Carcamo, Therese Michael, and Ryan Lucas for seamlessly handling all needed administrative work throughout my PhD years. Thanks are also due to J. Palmisano, H. Elsabagh, and A. Abdelkader for proofreading and commenting on some parts of this work.

I thank my friends for their understanding and sincere companionship that made my journey more bearable and exceptionally enriching.

I thank my extended family for their support and well-wishes. I am especially grateful to my uncles for their continuous encouragement; for supporting my education; and for providing me with my first programming CD set, cellphone, and laptop.

Last, but certainly not least, I thank my parents for exerting their capacity to provide for us; for forgiving much; and for putting up with much. I am deeply thankful for my Dad for raising me up with a love of engineering and for my first programming book; my Mom for her continuous prayers that enlighten my life; and my siblings for never giving up on me. I hope they all forgive me for being away. I thank my wife for her unfailing tolerance and for standing by me through thick and thin. Nothing would have been possible without my family's presence and encouragement. My appreciation to them is beyond words; may my Lord bestow His mercy upon them.

Mohamed E Elsabagh, 2017
# Table of Contents

<table>
<thead>
<tr>
<th>List of Tables</th>
<th>ix</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>xi</td>
</tr>
<tr>
<td>List of Abbreviations</td>
<td>xiii</td>
</tr>
<tr>
<td>Abstract</td>
<td>xiv</td>
</tr>
</tbody>
</table>

## 1 Introduction

1.1 Thesis Statement and Contributions | 5
1.2 What is Not Addressed | 7
1.3 Dissertation Organization | 8

## 2 Application-Level DoS: Background and Related Work

2.1 Application-Level Denial-of-Service Attacks | 9
2.2 Related Work
  2.2.1 Runtime Profiling | 10
  2.2.2 Failure Prediction | 11
  2.2.3 Formal Methods | 11
  2.2.4 Static Solutions | 12
  2.2.5 Syscall Tracing | 12

## 3 Early Detection of Application-Level Resource Exhaustion and Starvation

3.1 Assumptions and Threat Model | 13
3.2 Architecture of Radmin
  3.2.1 Kernel Tracer | 15
  3.2.2 User Tracer | 17
  3.2.3 Radmin Guard | 18
3.3 Learning and Detection
  3.3.1 Encoding | 20
  3.3.2 Learning the PFAs | 21
  3.3.3 Anomaly Detection | 24
3.4 Empirical Evaluation | 24
  3.4.1 Procedure and Metrics | 26
  3.4.2 Synthetic Exhaustion Attacks | 27
  3.4.3 Synthetic Attacks on Apache and W3m | 28
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1.3</td>
<td>Inheritance</td>
<td>109</td>
</tr>
<tr>
<td>8.2</td>
<td>Problem Definition</td>
<td>110</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Assumptions and Threat Model</td>
<td>110</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Vtable Attacks</td>
<td>111</td>
</tr>
<tr>
<td>8.2.3</td>
<td>Overview of VCI</td>
<td>112</td>
</tr>
<tr>
<td>8.3</td>
<td>Design and Implementation</td>
<td>113</td>
</tr>
<tr>
<td>8.3.1</td>
<td>Identifying Virtual Tables</td>
<td>113</td>
</tr>
<tr>
<td>8.3.2</td>
<td>Identifying Constructors</td>
<td>113</td>
</tr>
<tr>
<td>8.3.3</td>
<td>Inferring Class Layouts and Hierarchies</td>
<td>115</td>
</tr>
<tr>
<td>8.3.4</td>
<td>Identifying Virtual Calls</td>
<td>117</td>
</tr>
<tr>
<td>8.3.5</td>
<td>Class Type Propagation and Pairing</td>
<td>118</td>
</tr>
<tr>
<td>8.3.6</td>
<td>Policy Generation and Enforcement</td>
<td>121</td>
</tr>
<tr>
<td>8.4</td>
<td>Evaluation</td>
<td>122</td>
</tr>
<tr>
<td>8.4.1</td>
<td>Identification Accuracy</td>
<td>122</td>
</tr>
<tr>
<td>8.4.2</td>
<td>Security Effectiveness</td>
<td>125</td>
</tr>
<tr>
<td>8.4.3</td>
<td>Performance Overhead</td>
<td>130</td>
</tr>
<tr>
<td>8.5</td>
<td>Discussion and Improvements</td>
<td>131</td>
</tr>
<tr>
<td>8.6</td>
<td>ABI Dependency</td>
<td>131</td>
</tr>
<tr>
<td>8.7</td>
<td>Why not depend on RTTI?</td>
<td>131</td>
</tr>
<tr>
<td>8.7.1</td>
<td>Position-Independent Code (PIC)</td>
<td>132</td>
</tr>
<tr>
<td>8.7.2</td>
<td>Heterogeneous Containers</td>
<td>133</td>
</tr>
<tr>
<td>8.7.3</td>
<td>Virtual-dispatch-like C Calls</td>
<td>133</td>
</tr>
<tr>
<td>8.8</td>
<td>Destructors Corner Cases</td>
<td>134</td>
</tr>
<tr>
<td>8.9</td>
<td>Cross-module Polymorphism</td>
<td>135</td>
</tr>
<tr>
<td>8.10</td>
<td>Comparison to Related Policies</td>
<td>135</td>
</tr>
<tr>
<td>8.10.1</td>
<td>Reference Counts</td>
<td>135</td>
</tr>
<tr>
<td>8.10.2</td>
<td>Calling Convention</td>
<td>135</td>
</tr>
<tr>
<td>8.10.3</td>
<td>Call Arity</td>
<td>136</td>
</tr>
<tr>
<td>8.11</td>
<td>Summary</td>
<td>137</td>
</tr>
<tr>
<td>9</td>
<td>Concluding Remarks and Future Directions</td>
<td>138</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>141</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Checkpoint sites monitored by the tracing modules.</td>
<td>17</td>
</tr>
<tr>
<td>3.2 Hyperparameters used in training the PFAs.</td>
<td>26</td>
</tr>
<tr>
<td>3.3 Detection performance on Apache and W3m.</td>
<td>29</td>
</tr>
<tr>
<td>3.4 Detection performance for common Linux programs.</td>
<td>30</td>
</tr>
<tr>
<td>3.5 Starvation detection performance.</td>
<td>31</td>
</tr>
<tr>
<td>3.6 Apache Killer and Slowloris detection performance.</td>
<td>38</td>
</tr>
<tr>
<td>3.7 Checkpoint sites in user space.</td>
<td>39</td>
</tr>
<tr>
<td>3.8 Detection performance of URadmin.</td>
<td>41</td>
</tr>
<tr>
<td>3.9 Degree of vulnerability $D_{vuln}$ for each program used in our experiments</td>
<td>47</td>
</tr>
<tr>
<td>4.1 Kernel tracepoints hooked by Cogo for network I/O monitoring.</td>
<td>61</td>
</tr>
<tr>
<td>4.2 Summary of results for Apache.</td>
<td>64</td>
</tr>
<tr>
<td>4.3 Summary of OpenSIPS results.</td>
<td>69</td>
</tr>
<tr>
<td>6.1 Top 15 characteristics sorted by discrimination power.</td>
<td>84</td>
</tr>
<tr>
<td>6.2 Dataset used in the experiments.</td>
<td>89</td>
</tr>
<tr>
<td>8.1 Analysis result of SPEC CPU2006 and Firefox.</td>
<td>125</td>
</tr>
<tr>
<td>8.2 VCI policy coverage and average target reduction results.</td>
<td>127</td>
</tr>
<tr>
<td>8.3 Vcall target resolution statistics.</td>
<td>129</td>
</tr>
<tr>
<td>8.4 Percentage of fully, partially, and unresolved vcalls.</td>
<td>129</td>
</tr>
<tr>
<td>8.5 Performance overhead of VCI.</td>
<td>131</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Reported vulnerabilities by type (2010 – 2017)</td>
<td>2</td>
</tr>
<tr>
<td>3.1 Architecture of Radmin.</td>
<td>16</td>
</tr>
<tr>
<td>3.2 Overview of Radmin Guard</td>
<td>20</td>
</tr>
<tr>
<td>3.3 Example of a PST and the corresponding PFA over the alphabet $\Sigma = {a, b, c}$.</td>
<td>23</td>
</tr>
<tr>
<td>3.4 Runtime overhead incurred by Radmin.</td>
<td>32</td>
</tr>
<tr>
<td>3.5 Apache total maximum memory usage.</td>
<td>34</td>
</tr>
<tr>
<td>3.6 Apache total maximum open sockets.</td>
<td>35</td>
</tr>
<tr>
<td>3.7 Apache CPU usage per worker process.</td>
<td>36</td>
</tr>
<tr>
<td>3.8 Apache server availability without and with Radmin.</td>
<td>37</td>
</tr>
<tr>
<td>3.9 Runtime overhead incurred by URadmin.</td>
<td>42</td>
</tr>
<tr>
<td>4.1 Bounded GST and final PFA produced by Cogo.</td>
<td>58</td>
</tr>
<tr>
<td>4.2 Cogo’s architecture within Radmin.</td>
<td>61</td>
</tr>
<tr>
<td>4.3 HTTP DoS testbed used in the experiments.</td>
<td>62</td>
</tr>
<tr>
<td>4.4 Apache server availability against non-aggressive slow-rate attacks.</td>
<td>65</td>
</tr>
<tr>
<td>4.5 Apache server availability against aggressive slow-rate attacks.</td>
<td>66</td>
</tr>
<tr>
<td>4.6 SIP DDoS testbed used in the experiments.</td>
<td>68</td>
</tr>
<tr>
<td>4.7 Cogo’s detection of bye and invite floods against OpenSIPS.</td>
<td>70</td>
</tr>
<tr>
<td>5.1 Example of a ROP gadget.</td>
<td>72</td>
</tr>
<tr>
<td>6.1 Workflow of EigenROP.</td>
<td>80</td>
</tr>
<tr>
<td>6.2 Architecture of EigenROP within Pin.</td>
<td>88</td>
</tr>
<tr>
<td>6.3 Overall ROC of EigenROP.</td>
<td>91</td>
</tr>
<tr>
<td>6.4 AUC for different sampling intervals.</td>
<td>92</td>
</tr>
<tr>
<td>6.5 AUC with and without the microarchitecture-independent characteristics.</td>
<td>93</td>
</tr>
<tr>
<td>6.6 AUC for different sliding window sizes.</td>
<td>93</td>
</tr>
<tr>
<td>6.7 Overhead-accuracy tradeoff of EigenROP.</td>
<td>94</td>
</tr>
<tr>
<td>8.1 Sample C++ classes and vtables.</td>
<td>107</td>
</tr>
<tr>
<td>8.2 Assembly snippets for invoking $A::\text{bar}()$.</td>
<td>108</td>
</tr>
<tr>
<td>8.3 Assembly snippet for invoking $C::\text{foo}()$ using a base pointer</td>
<td>108</td>
</tr>
</tbody>
</table>
8.4 Overview of VCI. ................................................................. 112
8.5 Example C++ program and VCI's injected policy. ................. 123
8.6 Extracted PDG backward slice. ............................................ 124
8.7 Number of compatible target functions in libxul.so. ............... 137
## List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABI</td>
<td>Application Binary Interface</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ASLR</td>
<td>Address Space Layout Randomization</td>
</tr>
<tr>
<td>AV</td>
<td>Antivirus</td>
</tr>
<tr>
<td>CFG</td>
<td>Control Flow Graph</td>
</tr>
<tr>
<td>CFI</td>
<td>Control Flow Integrity</td>
</tr>
<tr>
<td>COOP</td>
<td>Counterfeit Object-Oriented Programming</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial Off-The-Shelf</td>
</tr>
<tr>
<td>CRA</td>
<td>Code-Reuse Attacks</td>
</tr>
<tr>
<td>DEP</td>
<td>Data Execution Prevention</td>
</tr>
<tr>
<td>DFI</td>
<td>Data Flow Integrity</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>DoS</td>
<td>Denial-of-Service</td>
</tr>
<tr>
<td>FN</td>
<td>False Negative</td>
</tr>
<tr>
<td>FP</td>
<td>False Positive</td>
</tr>
<tr>
<td>FPR</td>
<td>False Positive Rate</td>
</tr>
<tr>
<td>FRA</td>
<td>Function-Reuse Attacks</td>
</tr>
<tr>
<td>GCC</td>
<td>GNU Compiler Collection</td>
</tr>
<tr>
<td>GST</td>
<td>Generalized Suffix Tree</td>
</tr>
<tr>
<td>HPC</td>
<td>Hardware Performance Counters</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction Set Architecture</td>
</tr>
<tr>
<td>JOP</td>
<td>Jump-Oriented Programming</td>
</tr>
<tr>
<td>KPCA</td>
<td>Kernel Principle Component Analysis</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>NSID</td>
<td>Namespace Identifier</td>
</tr>
<tr>
<td>NSPID</td>
<td>Namespace Process Identifier</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PDG</td>
<td>Program Dependency Graph</td>
</tr>
<tr>
<td>PFA</td>
<td>Probabilistic Finite Automata</td>
</tr>
<tr>
<td>PIC</td>
<td>Position-Independent Code</td>
</tr>
<tr>
<td>PID</td>
<td>Process Identifier</td>
</tr>
<tr>
<td>PST</td>
<td>Probabilistic Suffix Tree</td>
</tr>
<tr>
<td>RASP</td>
<td>Runtime Application Self-Protection</td>
</tr>
<tr>
<td>RBF</td>
<td>Radial Basis Function</td>
</tr>
<tr>
<td>RCE</td>
<td>Remote Code Execution</td>
</tr>
<tr>
<td>RCG</td>
<td>Resource Control Graph</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computing</td>
</tr>
<tr>
<td>ROC</td>
<td>Receiver Operating Characteristics</td>
</tr>
<tr>
<td>ROP</td>
<td>Return-Oriented Programming</td>
</tr>
<tr>
<td>RTTI</td>
<td>Runtime Type Information</td>
</tr>
<tr>
<td>SIP</td>
<td>Session Initiation Protocol</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>TN</td>
<td>True Negative</td>
</tr>
<tr>
<td>TP</td>
<td>True Positive</td>
</tr>
<tr>
<td>TPR</td>
<td>True Positive Rate</td>
</tr>
<tr>
<td>TX/RX</td>
<td>Transmit/Receive</td>
</tr>
<tr>
<td>UA</td>
<td>User Agent</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>VoIP</td>
<td>Voice over IP</td>
</tr>
<tr>
<td>VTT</td>
<td>Virtual Tables Table</td>
</tr>
<tr>
<td>W-UA</td>
<td>Weaponized User Agent</td>
</tr>
</tbody>
</table>
Abstract

PROTECTION FROM WITHIN: RUNTIME HARDENING TECHNIQUES FOR COTS BINARIES

Mohamed Elsabagh, PhD
George Mason University, 2017
Dissertation Director: Dr. Angelos Stavrou

Software systems are the backbone of modern life as they drive all computerized technologies. The ever-increasing size and complexity of today's systems makes them very challenging to properly design and test, resulting in an incomprehensible attack surface that leads to poor — or altogether missing — security countermeasures. Owing to implementation and testing deficiencies, security defenses are deployed at the network and host perimeters to increase cyber coverage against attacks. Unfortunately, the current poor state of systems security bespeaks that perimeter security is not effective, especially with the proliferation of mobile, cloud, and IoT services where the perimeter cannot be clearly defined.

In this dissertation, I offer novel techniques to protect applications against attacks by retrofitting them with runtime shielding layers that defend from within the application itself rather than presuming that malicious inputs are blocked at the perimeter. These layers enable the application to detect and react to errant behavior by making it aware of its benign behavior and legitimate execution flow. I present automatic techniques that defend against the two most common classes of attacks on software systems: Denial-of-Service (DoS) attacks and Code-Reuse Attacks (CRA). The presented techniques directly embed security defenses into program binaries without requiring any side information such as source code, debug symbols, annotations, or attack signatures.
The first part of the dissertation presents Radmin and Cogo as two novel systems for early detection of resource DoS attacks at the application level. These are attacks that can result in program termination (crashing) by exploiting specific design and implementation weaknesses that force the program to over-consume or starve for resources. I present Probabilistic Finite Automata (PFA) based algorithms that save valuable system resources by detecting application-level DoS attacks in their early stages. I demonstrate and contrast the effectiveness of Radmin and Cogo on large-scale servers against comprehensive synthetic and real-world attacks.

The second part of the dissertation presents novel systems to defend against Return-Oriented Programming (ROP) attacks and Function-Reuse Attacks (FRA). I present EigenROP as a system for transparent detection of ROP attacks by monitoring runtime program characteristics such as memory locality and reuse distances. I then present VCI as a static binary rewriting system that retrofits C++ binaries with protections against FRA. EigenROP and VCI are significantly more precise than state-of-the-art binary solutions and can defeat sophisticated attacks.

The solutions presented in this dissertation raise defenses to a new level, making a strong case for automatic runtime hardening as a promising approach towards effectuating resilient systems that remain constructively functioning under attacks or even after sustaining some damage.
Software systems are often engineered and tested for functionality under *normal* rather than *adverse* conditions. This makes the systems vulnerable to attacks, where attackers engineer conditions that violate design and implementation assumptions. The result is a plethora of vulnerable, complex, systems that are impossible to redesign or reimplement from scratch. As the current security ecosystem depends on software vendors to provide security patches, end users are forced to either continue operating with little security assurance, or to shut the systems down till fixes are made available — a very poor trade-off between security and availability.

Current conventional defenses, such as intrusion detection systems and application firewalls, depend on static input and structural signatures distilled from attacks seen in-the-wild. This makes them incapable of keeping up against today's evolving cybersecurity threats. Attackers bypass these defenses by engineering new ways and new attack variants that do not match the preidentified signatures. After all, a defense that puts strong assumptions on the structure or internals of attacks is as good as its signatures database. Hence, it is arguable that current defenses fail to provide plausible protection against attacks.

Recent surveys have shown that only two vulnerability types account for more than 50% of all vulnerabilities and attacks reported between 2010 and 2017 [24, 37] (Figure 1.1) in Commercial Off-The-Shelf (COTS) software.\(^1\) The first is Denial-of-Service (DoS) vulnerabilities, where a program fails to properly restrict the amount of resources consumed or influenced by an attacker, leading to resource exhaustion or starvation and inability to serve benign clients. The second class is code execution vulnerabilities exploited via Code-Reuse Attacks (CRA), where existing executable code in the program memory is repurposed to execute arbitrary functionalities, without injecting any

\(^1\)COTS software are software tools and systems ready-made and available to the general public [1]. COTS can be purchased, leased, or freely licensed. For example, Microsoft Office, FireFox, Linux, are all COTS software.
new code. What makes matters worse is that multiple DoS and CRA attacks that were once thought purely theoretical are now a reality. For instance, it has been shown that CRA is turing-complete and works on RISC processors [53, 58]. And while network layer DoS defenses are widely deployed, attackers have moved up the protocol stack, and application layer DoS attacks are becoming a new norm [2, 24, 61].

DoS attacks targeting the network and transport layers have attracted considerable research attention [84, 89, 142]. Meanwhile, attacks have become more sophisticated and attackers have moved to higher layers of the protocol stack. Since 2010, resource exhaustion DoS attacks that operate at the application level have become more prevalent [2, 61] than attacks at the network and transport layers. Application-level exhaustion and starvation attacks, on the other hand, have received very little attention. Application-level DoS attacks pose a challenging and alarming threat. This is due to the following unique characteristics of the attacks: 1) They generally depend on legitimate-looking requests that pass through firewalls [2, 24, 61, 67]. 2) They often leverage inherent design weaknesses

---

**Figure 1.1:** Reported vulnerabilities by type (2010 – 2017).²

²Data courtesy of www.cvedetails.com [Retrieved 2017-06-01]. Percentages are rounded up, exploits were ignored, and vulnerability types with less than 3% share were also ignored.
of the target software [10, 61, 67]. 3) The attacks aim at reaching resource limits of individual programs rather than of the hardware or the infrastructure capabilities of the host system [10, 61]. 4) The attacks are inexpensive to deploy, requiring few resources from the attackers [61].

Current DoS detection solutions at the application level fall short of detecting or mitigating the attacks, mainly due to the following: 1) Static solutions [54, 59, 90, 136, 177] require source code access and annotations and only provide hints of the location of bottlenecks rather than providing fixes. 2) Dynamic solutions [43, 44, 62, 89, 122, 123, 143, 150, 151] enforce resource limits that are single point estimates, set irrespective of the actual consumption of different states of the target program for different inputs. This allows a big margin for attackers to exhaust resources by consuming the maximum amount of possible resources below the threshold. 3) Current solutions, due to their limited visibility into the application behavior, cannot detect exhaustion in its early stages. Even though the consumed resources may be very anomalous for the target program, detection only happens if the resource limits are crossed. 4) Setting only an upper limit on resource consumption allows starvation attacks to evade detection, since starvation involves not consuming resources.

It is important to realize that the scope of resource DoS attacks is not limited to availability. These attacks often cause collateral damage and carry other goals rather than simply consuming the available resources. For example, entropy sources may deplete and take longer to regenerate if excessively queried (e.g., [20]); devices may permanently brick when critical system components are forced to starve for resources (e.g., [102, 103]); side attacks may evade detection by exhausting system logging facilities (e.g., [135]); or systems may fail-open under resource exhaustion conditions permitting code execution attacks (e.g., [14, 32]). In fact, code execution and DoS are tightly coupled as it is not uncommon for reconnaissance and failed exploitation attempts to result in DoS [61, 67, 172].

One of the oldest code execution exploitation techniques is code injection: Malicious code is injected into the memory (data) of the target process and control flow is hijacked and redirected to the injected code. However, code injection has been mitigated by modern CPU and operating systems that guarantee W⊕X [39]. Memory pages cannot be both writeable and executable at the same time, limiting the possibility of code injection attacks.
But W+X is not infallible. One of the most common anti-W+X techniques is to launch Code-Reuse Attacks (CRA) [50], where existing code in the executable memory of the target process is reused to achieve arbitrary code execution. In a CRA, the attackers construct the payload by chaining instruction sequences (gadgets) from the executable process memory. The gadgets are chained using attacker-controlled data that influences the control flow such as return addresses on the stack and function pointers. This has yielded two very popular CRA techniques: 1) Return-Oriented Programming (ROP) [152], where unintended sequences of instructions are chained by returning or jumping to executable bytes using the attacker-controlled stack; and 2) Function-Reuse Attacks (FRA) [100, 133, 173], where whole functions are reused by overwriting function pointers. This is especially prevalent in programs written in Object-Oriented Programming (OOP) languages (for example, C++ and C#) that depend heavily on indirect calls (calls via function pointers) to provide polymorphism.

Over the past few years, research in ROP protection has become an arms race, where emerging defenses are countered by new subtle new variations of ROP attacks. ROP defenses can be defined in two major categories. The first category attempts to prevent ROP attacks at compile time by eliminating gadgets from binaries [117, 128] or enforcing Control-Flow Integrity (CFI) [40]. The second category aims to detect ROP attacks at runtime by monitoring the execution of programs [63, 70, 72, 121, 129, 159]. Unfortunately, existing ROP defenses suffer from shortcomings [58, 69, 88, 94, 170]: 1) They require access to source code and compiler support. 2) They focus on specific types of gadgets. 3) They depend on accurate disassembly and construction of Control Flow Graphs (CFGs). 4) They use hardware-dependent characteristics that are prone to evasion.

While the majority of code-reuse defenses focus on detecting ROP, FRA has received little attention. Primarily, this is because defending against FRA requires two complex operations: 1) distinguishing between legitimate and malicious calls of the same function, and 2) statically identifying virtual call targets, which requires recovering (at least partially) the semantics of the compiled program. Existing FRA defenses [100, 124, 133, 173] are far from complete or efficient, allowing attackers to reuse existing virtual function tables (tables of pointers to functions) to achieve code execution rather than directly reusing the functions themselves.

The brief sketch of the state of the art of DoS and CRA defenses offered here is discussed in depth in Chapters 2, 5 and 7.
1.1 Thesis Statement and Contributions

This dissertation makes a case for automatic runtime binary hardening as an effective and reliable countermeasure against DoS and CRA. I present several novel techniques that automatically retrofit applications and their runtime environments with protections that enable them to issue early warnings and defend against attacks. The proposed techniques work as runtime “shielding layers” that monitor and react to errant behavior, rather than relying solely on development decisions or perimeter security. These techniques work directly on program binaries\(^3\), without needing side information, such as source code and debug symbols. Binary-only solutions are highly desirable, since the source code of many programs — including commercial products, third party libraries, legacy software and firmware — is not available. Even if the source code is available, compiling new protections is not always feasible or desirable due to the presence of legacy code and compiler dependencies.

The techniques presented in this dissertation come under the umbrella of Runtime Application Self-Protection (RASP). Kephart and Chess [107] identified self-protection as one of the principal attributes needed for autonomic computing systems. They defined self-protection as a system’s ability to issue early warnings to anticipate and prevent system-wide failures, and to defend against malicious attacks or cascading failures. Gartner defines RASP [26] as security technology built or linked into an application or its runtime environment, capable of controlling application execution and preventing real-time attacks. The goal of RASP and the techniques presented herewith is to harden the application itself at runtime by making it aware of its benign behavior and able to detect unexpected behavioral changes and execution flows, rather than solely depending on perimeter security to block malicious inputs.

The novel techniques presented in this dissertation offer several key advantages over conventional and perimeter security. Perhaps the biggest of which is that there is no need to know where the vulnerabilities are located in an application or what inputs could exploit them. The runtime shields work as autonomic patches against vulnerabilities and zero-day attacks. Runtime hardening automatically diversifies defenses, moving away from the single point of failure oddity of conventional security defenses. By design, the proposed techniques cannot fail-open since the shielding layers

---

\(^3\)I use the term “binary” to refer to a compiled executable or a shared library. For the purpose of this dissertation, program, application, software, executable, and binary, are all synonyms unless otherwise explicitly stated.
always live within the hardened application throughout its life cycle. These techniques are suitable for handling arbitrary software without modification since they are application- and protocol-agnostic.

To summarize, the main contributions of this dissertation are the following:

- **Early detection of application-level DoS.** (Chapters 2 to 4) I propose Radmin [76, 78] as a novel system for early detection of application-level resource exhaustion and starvation attacks, i.e., before an attack wastes more resources than benign executions would typically consume. Radmin works directly on compiled binaries. It learns multiple Probabilistic Finite Automata (PFAs) from benign runs of target programs, then uses the PFAs as shadow state machines at runtime to confine resource consumption and detect resource DoS in its early stages. I demonstrate the effectiveness of Radmin by testing it over a variety of synthetic resource exhaustion and starvation weaknesses on commodity off-the-shelf software and in-the-wild attacks on the Apache server. I also analyze the attacker’s knowledge. Finally, I compare the accuracy and effectiveness of two different architectures: Radmin which works in both the user and kernel spaces, and URadmin which works solely in user space.

I also present Cogo as an alternative system for early DoS detection. Radmin employs quadratic training time, which is unpractical for defending large-scale programs. It also does not monitor network I/O, cannot attach to already running processes, and has been largely tested on synthetic attacks, among other technical limitations. Unlike Radmin and other prior solutions, Cogo builds behavioral models in linear time and monitors network I/O events. In many cases, it can block attacks far before they impact legitimate live sessions. I demonstrate the effectiveness and performance of Cogo using commercial-grade testbeds of two large and popular Internet services: Apache and the VoIP OpenSIPS servers. Cogo required less than 12 minutes of training time to achieve high accuracy (less than 0.0194% false positives rate) while detecting a wide range of resource exhaustion attacks less than 7 seconds into the attack. Finally, Cogo had only two to three percent per-session overhead.

- **Detection of ROP by singly monitoring the runtime characteristics of programs.** (Chapters 5 and 6) I introduce EigenROP [77], a system to detect ROP payloads based on unsupervised statistical learning of program characteristics. For the first time, I study the feasibility and
effectiveness of using microarchitecture-independent program characteristics — namely, memory locality, register traffic, and memory reuse distance — for detecting ROP. I propose a novel directional statistics-based algorithm to identify deviations from the expected program characteristics during execution. I also implement a dynamic instrumentation prototype of EigenROP using Intel Pin and measure it against in-the-wild ROP exploits and payloads generated by the ROP compiler ROPC. EigenROP works transparently to the protected program without requiring debug information, source code or disassembly. Overall, EigenROP achieved significantly higher accuracy than prior anomaly-based solutions. It detected the execution of the ROP gadget chains with 81% accuracy, 80% true positive rate, only 0.8% false positive rate, and incurred comparable overhead to similar Pin-based solutions.

- **Prevention of CRA on C++ programs by protecting the integrity of virtual function calls.**

  (Chapters 7 and 8) I present VCI [79], a binary rewriting system that secures C++ binaries against vtable attacks. VCI works directly on stripped binary files. It identifies and reconstructs various C++ semantics from the binary and constructs a strict CFI policy by resolving and pairing virtual function calls (vcalls) with precise sets of target classes. The policy is enforced by instrumenting checks into the binary at vcall sites. Experimental results on SPEC CPU2006 and Firefox show that VCI is significantly more precise than state-of-the-art binary solutions. Testing against the ground truth from the source-based defense GCC VTV, VCI achieved greater than 60% precision in most cases, accounting for at least 48% to 99% additional reduction in the attack surface over prior binary defenses. VCI incurred a 7.79% average runtime overhead which is comparable to the state of the art. In addition, I discuss how VCI defends against real-world attacks, and how it impacts advanced vtable reuse attacks such as COOP.

1.2 What is Not Addressed

I do not address (D)DoS attacks at the network, link, or the communication channel levels. Early detection of assertion failures and invalid memory accesses (segmentation faults) are also outside the scope of this work. In terms of CRA, this work handles ROP and virtual table based FRA in C++
binaries. Other CRA vectors such as C-style code pointer corruption and binaries compiled from polymorphic languages besides C++ are not addressed.

The presented techniques offer early detection and alarming; remediation strategies after detection are not the focus of this work. While my prototypes apply some mitigation policies — including restarting or terminating an attacked process — these policies may not be suitable for each and every application. Protocol-specific steps may be necessary to guarantee proper remediation. For example, it is necessary to invoke the proper protocol states to hang up malicious VoIP calls in order to properly recover from a VoIP based DoS attack on a SIP [140] server.

1.3 Dissertation Organization

Chapter 2 discusses related DoS defenses. Chapter 3 presents Radmin, an early detector of application level DoS. In Chapter 4, I present Cogo as an extension of Radmin that offers linear time training, network IO monitoring, process migration, among other practical features. I discuss ROP attacks and defenses in Chapter 5. In Chapter 6, I present EigenROP as a transparent ROP detection defense. I discuss CRA in Chapter 7 and present VCI for CFI enforcement against attacks on C++ binaries in Chapter 8. Finally, Chapter 9 offers concluding remarks and discusses future research directions.
Chapter 2
Application-Level DoS: Background and Related Work

2.1 Application-Level Denial-of-Service Attacks

Availability of services plays a major – if not the greatest – role in the survivability and success of businesses. Recent surveys [9,18] have shown that business managers and customers alike are willing to sacrifice security for convenience, preferring systems that are more often in an operable state to systems that may offer higher levels of security at the expense of more failures. Any disruption to the availability of a service directly translates into lost productivity and profit. Businesses invest in deploying redundant hardware and replicas to increase the availability of the services they offer. However, as software designers often overlook Saltzer-Schroeder's “conservative design” principle [144], software systems are often engineered and tested for functionality under normal scenarios rather than worst-case conditions. Taking advantage of this weakness, attackers engineer worst-case scenarios to over-consume needed resources (resource exhaustion) or to starve target processes of resources (resource starvation), effectively resulting in partial or complete DoS to legitimate users.

A system is exposed to resource exhaustion and starvation attacks if it fails to properly restrict the amount of resources used or influenced by an actor [10]. This includes, but is not limited to, infrastructure resources such as bandwidth and connection pools, and computational resources such as memory and CPU time. These attacks can operate at the network and transport layers [172] or at the application layer [61,67]. The asymmetric nature of communication protocols, design and coding mistakes, along with the prohibitive expense of tasks such as large database joins, exponential worst-case algorithms, extensive encryption, all contribute to resource exhaustion and starvation attacks susceptibility.
It is vital to realize that the scope of resource exhaustion and starvation attacks is not limited to availability. The attacks often cause collateral damage and carry other goals rather than simply consuming the resources available to a target. For example, systems may fail-open under starvation conditions [14, 32]; side attacks may evade detection by exhausting logging facilities [135], or entropy sources may deplete and generate weak secrets when excessively queried [20].

While DoS attacks targeting the network and transport layers have attracted considerable research attention [84, 89, 142], the attacks have become more sophisticated and attackers are rapidly moving to higher layers of the protocol stack [2, 61]. Attacks at the application level pose a challenging threat due to the following unique characteristics [10, 61, 67]: First, these attacks depend on small, seemingly legitimate inputs that are able to pass through firewalls. Second, they leverage inherent weaknesses in the design of target programs, rather than violating security properties or exploiting programming bugs. Third, the attacks are inexpensive to deploy, requiring few resources from the attackers. Finally, it is challenging for developers to properly restrict the resources consumed by programs, since resource consumption has both code and input dependencies.

2.2 Related Work

Modern operating systems offer threshold-based facilities to limit the resource consumption of processes (e.g., setrlimit, ulimit, AppArmor). These facilities, while widely available, often fall short in detecting or mitigating resource exhaustion and starvation attacks. Their limits are set irrespective of the actual consumption of different program segments for different inputs or users. This enables attackers to exhaust resources by crafting high-consumption inputs that run for prolonged time periods [61, 67, 109]. In addition, these facilities cannot detect starvation attacks, since only an upper bound is specified.

2.2.1 Runtime Profiling

Several facilities exist for monitoring the performance of running processes (e.g., getrusage, proc/pid/stat). However, due to the small time granularity of execution, querying those facilities often yields inaccurate measurements that are not suitable for fine grained monitoring [44].
addition to performance monitoring, several static and dynamic instrumentation tools exist for profiling, such as gprof, Valgrind and the more generic, Intel Pin. However, the instrumentation overhead is often too high [141,162] to enable their continuous usage, especially when exhaustion detection is the goal. In [122], Matias et al. proposed a differential software analysis system that automatically detected software aging. Their approach employed two time series of several memory usage statistics, one for a given software program and another for a more robust version. By comparing the time series, software aging can be assumed if significant divergence is detected.

2.2.2 Failure Prediction

Several techniques have been proposed to predict the time to a complete resource depletion [143]. Cheng et al. [62] offered a failure prediction system that employs a health index $\in [0,1]$ and a linear regression function to estimate the mean time to exhaustion if the health index drops below a preset threshold. The choice of the regression polynomial required prior knowledge of the usage patterns. Also, the system, which required manual calibration, modeled the total amounts of resources used by a server machine, rather than program-specific behavior. This is particularly important because web applications depend on host system applications and libraries to complete common tasks, such as sending emails, filtering input, changing hardware settings, and running custom scripts. Therefore, resources consumed by individual programs and applications running on the system need be monitored or attacks can evade detection.

2.2.3 Formal Methods

Antunes et al. [44] proposed a system for testing server programs for exhaustion vulnerabilities. The system depended on user supplied specifications of the server protocol. It automatically generated (fuzzed) test cases and launched them against the server. The authors used ptrace to attach to the server process and recorded the performance stats from /proc. A regression model was used on the recorded measurements to build projections of the resource usage. In [89], Groza et al. formalized DoS attacks using a set of cost-based rules. A case study was provided using the Internet Key Exchange (IKE) and Just Fast Keying (JFK) protocols. Aiello et al. [43] formalized a set of protocol specifications
for establishing DoS resiliency, that, while promising, required explicit cost calculation of required computational resources — a requirement often not feasible in practice [172].

### 2.2.4 Static Solutions

Chang et al. [59] proposed a static analysis system was for identifying source code sites that may allow uncontrolled CPU time and stack consumption. The authors used taint and control-dependency analysis to automatically identify high complexity source code control structures that can be influenced by untrusted input. Similarly, approaches that require manual code annotation were developed, including [90, 136, 177]. Burnim et al. [54] used symbolic execution to generate inputs that exhibit worst-case time complexity. However, symbolic execution cannot scale to handle real-world applications, and the need for side information, such as source code annotations and debug symbols, is not applicable in many scenarios (e.g., commercial and closed source applications). Additionally, exhaustion is not limited to CPU time and stack size; it can still happen through other vectors, such as exhausting file descriptors and thread pools.

### 2.2.5 Syscall Tracing

In [150,151], Sekar et al. introduced approaches for detecting abnormal program behavior by building automata from system calls and executing the automata at runtime, flagging invalid transitions as anomalies. Mazeroff et al. [123] described methods for inferring and using probabilistic models for detecting anomalous sequences of system calls. They built a baseline model of system call sequences executed by benign programs, along with a test model of a target program, and compared the distance between the two models to detect anomalies. While approaches based on system call monitoring are easy to deploy, they are prone to mimicry attacks [106, 130]. Additionally, they either completely ignore call arguments, which makes them inapplicable for exhaustion detection; or they model the arguments using point estimates, which is insufficient for early exhaustion detection.
Chapter 3
Early Detection of Application-Level Resource
Exhaustion and Starvation

In this chapter, I present Radmin [76, 78], a system for automatic early detection of application-level resource exhaustion and starvation attacks. By application-level attacks, I refer to the classes of DoS attacks that utilize small, specially crafted malicious inputs that cause uncontrolled resource consumption in victim applications. To this end, Radmin traces the resource consumption of a target program in both the user and kernel spaces (see Section 3.2) and builds and executes multiple state machines that model the consumption of the target program.

The key observation is that attacks result in abnormal sequences of transitions between the different resource consumption levels of a program when compared to normal conditions. By modeling the resource consumption levels as multiple realizations of a random variable, conditional distribution of the current consumption level given the history (context) of measurements can be estimated. Consequently, the statistical properties of the resulting stochastic process can be used to detect anomalous sequences.1

Radmin operates in two phases: offline and online. In the offline phase, the monitored programs are executed on benign inputs, and Radmin builds multiple Probabilistic Finite Automata (PFA) models that capture the temporal and spatial information in the measurements. The PFA model is a finite state machine model with a probabilistic transition function (see Section 3.3). Both the time of holding a resource and the amount of resource used are mapped to states in the PFA, while changes in the states over the time are mapped to transitions.

In the online phase, Radmin executes the PFAs as shadow resource consumption state machines, using the transition probabilities from the PFAs to detect anomalous consumption. Additionally,

1Unless stated otherwise, I use “measurements” and “sequences” interchangeably in the rest of this chapter.
Radmin uses a heartbeat signal to time out transitions of the PFAs. Together with the transition probabilities, this enables Radmin to detect both exhaustion and starvation attacks.

Radmin aims at detecting attacks as early as possible, i.e., before resources are wasted either due to exhaustion or starvation. Radmin does not use any static resource consumption thresholds. Instead, the PFAs capture the transitions between the different consumption levels of different program states, and statistics of the PFAs are used to detect anomalies. The PFAs allow Radmin to implicitly map different program states, i.e., program behavior at some execution point given some input, to dynamic upper and lower resource consumption bounds.

For the purpose of this dissertation, earliness of detection is quantified as the ratio of resources that Radmin can save as compared to the maximum amounts of resources consumed in benign conditions (see Section 3.4). This corresponds to the tightest static threshold that traditional defenses can set, without causing false alarms. Radmin has an advantage over all existing defenses that use static thresholds (see Chapter 2), since exhaustion and starvation attacks can evade those defenses. Exhaustion attacks can consume the highest amounts of resources possible, just below the static threshold [2, 61]. Additionally, starvation attacks, by design, do not aim at consuming resources [61].

In summary, this chapter makes the following contributions:

1. Demonstrates that Radmin can detect both resource exhaustion and starvation attacks in their early stages by employing a novel detection algorithm that uses PFAs and a heartbeat signal. Radmin takes both temporal and spatial resource consumption information into account and adds minimal overhead.

2. Implements a prototype that uses kernel event tracing and user space instrumentation to efficiently and accurately monitor resource consumption of target processes.

3. Demonstrates the effectiveness of Radmin using a broad range of synthetic attacks against common Linux programs, showing that Radmin can efficiently detect both types of anomalies in their early stages with low overhead and high accuracy.

4. Demonstrates the effectiveness of Radmin using two common in-the-wild DoS attacks against Apache, namely Apache Killer [6] and Slowloris [30], illustrating that Radmin can efficiently and accurately detect both attacks in their early stages.
5. Implements a new prototype, URadmin, that operates solely in user-space. Uradmin is easier to deploy by end users without requiring kernel tracing.

6. Compares the accuracy and overhead of Radmin to URadmin.

7. Proposes a metric to measure the degree of vulnerability of a program to resource exhaustion that may be triggered using benign inputs that conform to the PFAs enforced by Radmin.

### 3.1 Assumptions and Threat Model

Radmin’s main goal is early detection of application-level resource exhaustion and starvation, which may result in full or partial depletion of available resources (CPU time, memory, file descriptors, threads, and processes) or starvation and stalling. I assume that actors can be local or remote, with no privilege to overwrite system binaries or modify the kernel.

I consider the following types of exhaustion and starvation attacks. First, attacks that result in a sudden surprisingly high or low consumption of resources (e.g., an attacker controlled value passed to a `malloc` call). Second, attacks that result in atypical resource consumption sequences such as algorithmic and protocol-specific attacks that aim at maximizing (flattening) the amounts of consumed resources. Third, attacks that cause stalling of execution, including triggering livelocks or prolonged locking of resources.

In my experiments, although I considered only programs running on x86 Linux systems and following the Executable and Linkable Format (ELF), the proposed approach places no restrictions on the microarchitecture, the binary format, or the runtime environment.

### 3.2 Architecture of Radmin

The major components of Radmin are a kernel space tracing module (Kernel Tracer), a user space tracing library (User Tracer), and a daemon process (Guard) where the bulk of processing takes place. The tracing modules monitor and control a target program by binding checkpoints to events of interest, in the execution context of the target. Checkpoints are functions in the tracing modules that are called when an event of interest is triggered. Each checkpoint communicates measurements
Figure 3.1: Architecture of Radmin. The User Tracer and Kernel Tracer monitor and collect measurements from a target program by binding checkpoints to events of interest in both the user and kernel spaces. They send the measurements to the Guard, where the bulk of processing takes places.

and control information to the Guard. I refer to a code site at which an event was triggered as a checkpoint site. Figure 3.1 shows the system architecture of Radmin.

Radmin takes a target program binary as input, and operates in two phases: offline and online. In the offline phase, Radmin instruments the target binary by injecting calls to the User Tracer into the binary, and writes the instrumented binary to disk. Then, the instrumented program is executed over benign inputs while Radmin monitors its execution in both the user and kernel spaces using the User Tracer and the Kernel Tracer modules, respectively. During this stage, the Guard receives the measurements from the tracers and constructs multiple PFAs that capture the resource consumption behavior of the target program. Finally, in the online phase, the Guard executes the PFAs along with the target program and raises an alarm if a deviation of the normal behavior is detected (see Section 3.3).

Each measurement is a vector of \((\text{consumed kernel time}, \text{consumed user time}, \text{consumed resource amount})\) associated with a resource type and a task\(^2\) ID. (Here, “consumed resource amount” accounts for the total amount of a resource that would be in consumption if the allocation or deallocation request is granted.) Parent-child task relationships are tracked by recording both the parent and current task IDs, in addition to the process ID. The measurement vectors accurately capture the resource consumption behavior of a process, as they map out both the sequences of resource

\(^2\)Unless stated otherwise, I use “task” to refer to child processes and threads spawned by a monitored program.
Table 3.1: Checkpoint sites monitored by the tracing modules. Checkpoint sites used by the Kernel Tracer are given in the SystemTap probes notation.

<table>
<thead>
<tr>
<th>Checkpoint Site</th>
<th>User/Kernel</th>
<th>Resource Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>vm.brk</td>
<td>Kernel</td>
<td>Memory</td>
</tr>
<tr>
<td>vm.mmap, syscall.mmap2</td>
<td>Kernel</td>
<td>Memory</td>
</tr>
<tr>
<td>vm.munmap</td>
<td>Kernel</td>
<td>Memory</td>
</tr>
<tr>
<td>kernel.__fd_install</td>
<td>Kernel</td>
<td>File descriptors</td>
</tr>
<tr>
<td>kernel.__close_fd</td>
<td>Kernel</td>
<td>File descriptors</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Recursive sites</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sites that manipulate the stack pointer</td>
<td>User</td>
<td>Stack</td>
</tr>
<tr>
<td>scheduler.ctaswitch</td>
<td>Kernel</td>
<td>Stack</td>
</tr>
<tr>
<td>perf.av.cpu_clock</td>
<td>Kernel</td>
<td>Stack</td>
</tr>
<tr>
<td>Heartbeat every 500 ms.</td>
<td>Both</td>
<td>Stack</td>
</tr>
<tr>
<td>scheduler.wakeup_new</td>
<td>Kernel</td>
<td>Stack</td>
</tr>
<tr>
<td>kprocess.exec_complete</td>
<td>Kernel</td>
<td>Stack</td>
</tr>
<tr>
<td>kprocess.exit</td>
<td>Kernel</td>
<td>Stack</td>
</tr>
</tbody>
</table>

consumption changes and the time for each change, which effectively captures both the temporal and spacial information in the resource consumption behavior of the process.

I developed the user space components in C/C++, using Dyninst [11] for static binary rewriting. The kernel tracer was developed using SystemTap [33]. Coordination scripts and a command line interface also were developed in Shell Script. Radmin uses SystemTap to implement the kernel tracer in the high-level SystemTap syntax, where SystemTap compiles the tracer into a kernel module which Radmin uses directly at runtime. At compile-time, SystemTap is needed to compile the kernel tracer into a kernel module binary. However, it is important to note that this dependency is compile-time only. At runtime, only the staprun executable is needed to interact with the kernel module. Staprun is a regular program that does not add any specific restrictions on the runtime environment nor does it require SystemTap to be present, It can be bundled with Radmin.

A summary of the checkpoint sites and the associated resource types is shown in Table 3.1 and discussed in the following sections.

3.2.1 Kernel Tracer

The Kernel Tracer binds checkpoints to various kernel events by binding probes to the corresponding kernel tracepoints. Kernel tracepoints provide hooks to various points in the kernel code by calling functions (probes) that are provided at runtime by kernel modules [73]. Binding to the centralized,
well-defined, kernel tracepoints associated with resource (de)allocation is more robust than attempting to enumerate and trace, from user space, all possible ways a program can (de)allocate resources through library calls. Additionally, kernel tracing gives maximum visibility into the target process and allows for low-penalty monitoring and control of the target.

The Kernel Tracer keeps track of task creation by binding to the kernel scheduler wakeup tracepoint (scheduler.wakeup_neu), which is triggered when a task is being scheduled for the first time. It monitors task destruction by binding to the task exit tracepoint (kprocess.exit). The tracer also monitors processes overlaid by the exec call family by binding to the exec completion tracepoint (kprocess.exec_complete).

For memory monitoring, the Kernel Tracer install probes for the tracepoints that are triggered upon the allocation of contiguous memory (vm.brk), memory regions (vm.mmap), and the release of memory to the kernel (vm.munmap). In addition, the syscall.mmap2 probe point is used to cover architectures that use mmap2 instead of mmap [17]. Note that libc wrappers (e.g., malloc, calloc, free, etc.) internally call mmap and munmap to allocate and release memory. For file monitoring, probes are installed for kernel functions that update the allocated file descriptors in the file descriptors table of the process, namely: (kernel.__fd_install) and (kernel.__close_fd). For CPU monitoring, the Kernel Tracer keeps track of the consumed clock ticks by binding to the scheduler tracepoints that trigger when monitored tasks context switch (scheduler.ctxswitch) and when the kernel clock ticks (perf.sw.cpu_clock) inside the context of a monitored task. The reason for monitoring only these two events is to minimize the overhead of profiling CPU time.

It is important to note that even though memory is monitored from the kernel module, user space processes can exhaust their stack space without interfacing with the kernel. Therefore, additional checkpoints for monitoring the stack in user space were included.

### 3.2.2 User Tracer

The User Tracer consists of a user space library where calls to that library are injected in the target binary at assembly sites of interest. The User Tracer is injected as follows. First, Radmin statically parses the input binary and extracts a Control Flow Graph (CFG) using the Dyninst ParseAPI library. It then analyzes the CFG to identify assembly sites that dynamically operate on the stack such as
recursive calls (direct and indirect) and variable length arrays. Radmin injects calls to the tracer library at the marked sites in the binary and saves the modified binary to disk.

To calculate the stack size consumed by recursive call sites, two options were tested: 1) parsing the process memory maps from \texttt{/proc/<pid>/smaps}, and 2) unwinding the stack. Both options proved unreliable. The obtained values from \texttt{smaps} were too coarse to reflect actual stack consumption. Unwinding the stack was very expensive and required special arrangements at compilation time (e.g., the usage of frame pointers) that were not feasible to attain when working directly with compiled programs. Instead, Radmin implements a workaround by tagging (marking) the stack inside the caller function site at a point directly before the recursive call, and then calculating the distance from the entry point of the recursive callee function site to the tag. The tag is injected only in non-recursive caller function sites, avoiding mistaken tag overwriting due to indirect recursion.

Additionally, the User Tracer spawns a heartbeat thread that periodically consumes one clock tick then switches out. Consequently, the heartbeat tick is captured by the Kernel Tracer whenever the heartbeat thread schedules out. It delivers a clock signal from the monitored process to the Guard, which detects starvation attacks by testing if the transitions between the PFA states had timed out (see Section 3.3).

### 3.2.3 Radmin Guard

Figure 3.2 shows the underlying architecture of the Guard. In the offline phase, the Guard learns a codebook over a finite alphabet $\Sigma$, and encodes the incoming measurements over $\Sigma$. Encoding the measurements serves two purposes: 1) It discretizes the continuous measurements, making them useful for estimating the conditional probabilities using the PFAs; and 2) It reduces the dimensionality (lossy compression) of the measurements by mapping them to a finite alphabet of a much smaller size. The Guard then builds multiple PFAs over the encoded sequences, one for each monitored resource type. In the online phase, the Guard encodes the incoming stream of measurements, executes the PFAs (per task, per resource type), runs the detection algorithm, and raises an alarm if an anomaly is detected. In my experiments, I only terminated the violating process. However, more advanced recovery can be used such as resource throttling or execution rollback [154].
In the following section, I discuss in more depth how the Guard encodes the measurements, learns and executes the PFAs, and detects attacks.

### 3.3 Learning and Detection

#### 3.3.1 Encoding

Radmin learns each codebook used by the encoder by running a $k$-means quantizer over the raw vectors of measurements, where $k = |\Sigma|$ is the number of desired codewords. In my implementation, I used $k$-means++ [36, 45], which is guaranteed to find a codebook (clusters) that is $O(\log k)$-competitive with the optimal $k$-means solution [45]. To build the codebook, each measurement (consumed kernel and user time, resource value) is treated as a point in a three-dimensional space. $k$-means++ starts by selecting one center point at random from among all measurement points. Then, the distance $d(x)$ between each measurement point $x$ and the nearest center point is computed. Next, one more center point is chosen with probability proportional to $d^2(x)$. This seeding process repeats until $k$ centers are chosen. After which, standard $k$-means clustering is performed resulting in $k$ point clusters, the centers of which are the codewords. I refer the interested reader to [45] for a detailed discussion of $k$-means++.

Each codebook $\Sigma$ (one codebook per resource type) stores an indexed list of codewords. Each codeword $\sigma$ is represented by three-dimensional centers $\mu_\sigma$ and spreads $s_\sigma$, where each dimension corresponds to one dimension of the raw measurement vector. The number $|\Sigma|$ of codewords is determined such that each dimension gets at least 1 degree of freedom (level), constrained by a
total of 64 degrees of freedom per codeword, i.e., $|\Sigma| \in [3 \ldots 64]$. This setup allows at most four
degrees of freedom per dimension ($4^3$ total), in the case that all dimensions have the same amount
of variance. Finally, encoding is done by mapping a given measurement vector to the index of its
nearest codeword. If a measurement vector falls outside the coverage of all codewords, an empty
codeword $\emptyset$ is returned.

### 3.3.2 Learning the PFAs

Radmin builds PFAs for the monitored resource types and uses them to predict the probability of
new sequences of measurements given the history of measurements. A PFA is a 5-tuple $(\Sigma, Q, \pi, \tau, \gamma)$,
where:

- $\Sigma$ is a finite alphabet (the codebook) of symbols processed by the PFA.
- $Q$ is a finite set of PFA states.
- $\pi: Q \to [0, 1]$ is the probability distribution vector over the start states.
- $\tau: Q \times \Sigma \to Q$ is the state transition function.
- $\gamma: Q \times \Sigma \to [0, 1]$ is the emitted probability function when making a transition.

The subclass of PFA used in Radmin is constructed from their equivalent Probabilistic Suffix Tree
(PST) model [139], which is a bounded variable-order Markov model where the history length varies
based on the context (statistical information) of the subsequences of measurements, and the tree
does not grow beyond a given depth $L$. In other words, the PST captures all statistically significant
paths between resource consumption levels (encoded measurements), where the path length is at
most $L$. In the construction of the PST, a subsequence of encoded measurements $s \in \Sigma^*$ is added to
the PST only if:

1. $s$ has a significant prediction probability, i.e., there is some symbol $\sigma \in \Sigma$ such that $P(\sigma|s) \geq$
   $\gamma_{\text{min}}$, where $\gamma_{\text{min}}$ is the minimum prediction probability of the model.

2. And, $s$ makes a contribution, i.e., the prediction probability is significantly different from the
   probability of observing $\sigma$ after the parent node of $s$, i.e.,
   $\frac{P(\sigma|s)}{P(\sigma|\text{Parent}(s))} \geq r_{\text{min}}$ or $\leq \frac{1}{r_{\text{min}}}$, where
   $r_{\text{min}}$ is the minimum difference ratio.
The PFA model provides tight time and space guarantees since it has a bounded order, and only the current state and the transition symbol determine the next state. Those are desirable properties for Radmin since 1) I construct the PFAs without prior knowledge of the dependencies order (the length of statistical history in the measurements produced by target programs); and 2) I want to minimize the execution overhead of Radmin by maintaining a minimal amount of state-keeping information for the PFAs and calculating the prediction probability for each measurement as quickly as possible.

For a sequence of $n$ measurements, the PFA model allows us to compute the prediction probability in $O(n)$ time and $O(1)$ space. I refer interested readers to [48, 71, 139] for detailed discussions of various construction algorithms. In my prototype, I used the algorithm in [139], which builds the PFA in $O(n^2 L)$.

In the subclass of PFA used in Radmin, each state $q_i \in Q$ has a unique ID corresponding to the subsequence captured by that state, and the PFA has a single start state $q_o$, where $\pi(q_o) = 1$. Given a PFA $M$ and a string of encoded measurements $s = s_1 \ldots s_l$, we walk $M$ (for each $s_i \in s$) where each transition $q_{i+1} = \tau(q_i, s_i)$ emits the transition probability $\gamma(q_i, s_i)$. The prediction probability of $s$ by $M$ is given by:

$$P(s) = \prod_{i=1}^{l} \gamma(q_{i-1}, s_i).$$  \hspace{1cm} (3.1)

For example, given the PFA in Figure 3.3, the prediction probability of the sequence of encoded measurements “abca” is given by:

$$P(abca) = \gamma(\phi, a) \times \gamma(a, b) \times \gamma(ab, c) \times \gamma(c, a)$$  \hspace{1cm} (3.2)

$$= 3/8 \times 2/3 \times 1 \times 1/2$$  \hspace{1cm} (3.3)

$$= 1/8.$$  \hspace{1cm} (3.4)

Learning the PFAs for a target program requires running the target program over benign inputs. The following are some possible ways to handle this:

- **Dry runs and collected benign traffic.** Radmin can be trained through dry runs over benign inputs. This is typical in internal acceptance and pre-release testing. Radmin can also be trained
using traffic that has already been processed by applications and shown to be benign. This is arguably the easiest approach to train Radmin if it is deployed to protect a web-server.

- **Functionality tests.** Radmin can be trained using positive functionality tests. Testing is integral to the software development lifecycle, and Radmin can integrate with the test harness at development time. The main disadvantage is the additional effort needed for integration and debugging.

- **End users.** Radmin can be trained by end users. Even though this causes an increased risk of learning bad behavior, the resulting PFAs can be compared or averaged based on the type and privileges given to each class of users. The PFAs can be averaged, for example, based on the distance between their transition functions. Additionally, the learning algorithm can be modified such that the PFA learns new behavior if the new behavior is statistically similar to old behavior, by using statistics over the frequency of minimum probability transitions.

Once trained, Radmin can continue learning or be locked down based on the system policy. For example, system administrators may desire to limit guest users to what Radmin already knows, while PFAs for sudoers can adjust and add to what they learned. The PFAs also can be locked after some time of no change, which can be an effective strategy for preventing future attacks from compromised users. I discuss ways to avoid false positives and improve the PFA prediction performance in Section 3.8.1.
3.3.3 Anomaly Detection

In the online phase, the Guard operates by encoding the received sequences over $\Sigma$, and executing the corresponding PFAs as shadow automata, where each sequence results in a transition in one or more PFA. In addition to the measurements, the Guard uses the received heartbeat signal $t$ to timeout the transitions of the PFAs.

Algorithm 3.1 outlines the detection algorithm. Radmin raises an alarm if any of the following conditions is satisfied:

1. A foreign symbol is detected (lines 1–3). In this case, the program is requesting some resource amount that is not within the spread of any of the codewords in the codebook. This typically indicates an overshoot or undershoot signal. A common example is DoS attacks that use data poisoning to pass a huge value to a `malloc` call, resulting in immediate crashing.

2. The program is requesting a transition that has a very low probability (lines 4–5). This case captures scenarios where attackers craft input that consumes (or locks) resources at program states that differ from benign runs. A common example is an attack that aims to maximize the amount of resources consumed by the program.

3. One or more PFAs time out (lines 6–12). In this case, the program has not transitioned to any of the next states within an acceptable time, with respect to one or more resource types. This, for example, could indicate the presence of a livelock.

The algorithm takes $O(|\Sigma|)$ time in the worst case, since the number of outgoing edges from any state is at most $|\Sigma|$.

3.4 Empirical Evaluation

I conducted a series of experiments to evaluate the effectiveness of Radmin. The first experiment used a web server and a browser with sufficient input coverage. The second experiment used common Linux programs and the functionality tests that shipped with them as a representation of normal inputs. Finally, a third experiment evaluated the effectiveness of Radmin in detecting starvation, using starvation and livelock cases that are common in the literature.
Algorithm 3.1: Accept Measurement

\begin{algorithm}
\begin{algorithmic}
\State \textbf{input} : Measurement vector $v$
\hspace{1cm} Heartbeat $t$
\hspace{1cm} PFA $M$
\hspace{1cm} Current state $q_i \in Q$
\hspace{1cm} Current path probability $p$
\State \textbf{output} : Accept or Reject
\State $c \leftarrow \text{Encode}(v)$
\State \textbf{if} $c = \emptyset$ \textbf{then}
\State \hspace{1cm} \textbf{Reject} \quad \triangleright \text{Foreign value}
\State \textbf{if} $p \cdot \gamma(q_i, c) < \gamma_{\text{min}}(M)$ \textbf{then}
\State \hspace{1cm} \textbf{Reject} \quad \triangleright \text{Low probability transition or path}
\State \textbf{timedout} $\leftarrow$ \text{True}
\State \textbf{foreach} transition $\tau(q_i, s_i)$ \textbf{do}
\State \hspace{1cm} \textbf{if} $\neg \text{Timedout}(s_i, t)$ \textbf{then}
\State \hspace{2cm} timedout $\leftarrow$ \text{False}
\State \hspace{1cm} \textbf{break}
\State \textbf{if} timedout $= \text{True}$ \textbf{then}
\State \hspace{1cm} \textbf{Reject} \quad \triangleright \text{All transitions timed out}
\State \textbf{Accept} \quad \triangleright \text{take the transition}
\end{algorithmic}
\end{algorithm}

- $\gamma_{\text{min}}(M)$ is the minimum prediction probability of $M$.
- $\text{Timedout}(s_i, t)$ tests if the time signal $t$ lies outside the spread of the time dimensions of the codeword $s_i$ of transition $\tau(q_i, s_i)$. 
I refer to test cases that trigger abnormal behavior as *positive* (malicious), and those that do not as *negative* (benign). Each positive test case can either be detected as malicious or misdetected as benign, giving a true positive (TP) or a false negative (FN), respectively. Each negative test case can either be detected as benign or incorrectly detected as malicious, giving a true negative (TN) or a false positive (FP), respectively.

### 3.4.1 Procedure and Metrics

For every target program, I executed two thirds of the negative test cases to collect benign measurements and build the PFAs. Then, I executed the remaining one third to measure the false positive rate. Finally, I executed all positive test cases to measure the detection rate and earliness of the detection.

The PFAs were trained and optimized using 5-fold cross-validation (CV) over the training sequences (measurements from the two-thirds of negative test cases used in training). For each resource type, Radmin built one PFA and selected its hyperparameters from a cross product of all possible values (Table 3.2). Training sequences were divided into five roughly equal segments. Each fold in the CV used the sequences in one such segment for testing and a concatenation of the rest for training. CV testing was performed by calculating the average log-loss of the prediction probability of sequences, given by $-\frac{1}{T} \sum_{i=1}^{T} \log P(s_i)$, where $P(s_i)$ is the prediction probability of test sequence $s_i$ and $T$ is the total number of test sequences. This was done for each fold, resulting in five average log-loss values per hyperparameters vector. Finally, the hyperparameter vector with the best median log-loss over the five folds was used to build the PFA over the entire training sequences.

I used the following metrics in this evaluation: False Positive Rate (FPR), True Positive Rate (TPR), and Earliness (Erl.). Earliness was calculated as the percentage of the amount of resources that Radmin saved under an attack, to the maximum resources used by negative runs. I used Erl. to quantify how quickly Radmin detected the attacks. For example, if a program consumed a maximum of 40

### Table 3.2: Hyperparameters used in training the PFAs.

<table>
<thead>
<tr>
<th>Param. Possible values</th>
<th>Chosen (median)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\gamma_{min}$ {10$^{-9}$, 10$^{-7}$, 10$^{-11}$, 10$^{-13}$}</td>
<td>10$^{-11}$</td>
</tr>
<tr>
<td>$r_{min}$ {1.05}</td>
<td>1.05</td>
</tr>
<tr>
<td>$L$ {30, 40, 50, 60}</td>
<td>40</td>
</tr>
</tbody>
</table>
MB under benign conditions, and an attack consumed 30 MB before Radmin detection, the earliness of detecting the attack would be $\frac{40-30}{40} = 25\%$. Erl. reaches its best value at 100 and its worst at 0.

For resource exhaustion detection, I used synthetic attacks (discussed in the following section). In the case of starvation and livelocks, I used a number of common cases that appeared in prior livelock detection studies [23, 31, 85, 118]. Since the attacks aimed to exhaust system resources, they were always detected once consumed resources more than the maximum of benign runs. Therefore, Radmin always achieved a TPR of 100%. The same applies to starvation and livelock test cases.

### 3.4.2 Synthetic Exhaustion Attacks

One approach to evaluating Radmin against resource exhaustion attacks would be to test it with several known attacks. While such an approach is common in the literature, it suffers from two major drawbacks. First, it is challenging to identify real exhaustion attacks that exploit different weaknesses and resource types and exercise different code paths for each target program. Results can be biased because the number of attacks would have little to no correlation with the variety of attacks that can be detected. Second, evaluating a defense system against known attacks limits the scope of the evaluation and the results, and may establish a false sense of security [58, 106, 130]. Therefore, I decided against using only the few known attacks and instead opted for generating synthetic attacks that resembled attacks seen in the wild. The goal was to stress the system and determine its limits on a much richer set of attack entries.

To achieve this, I assumed that the attacker had successfully identified some exhaustion vulnerability in the target program, and had crafted malicious input that successfully triggers the vulnerability. The nature of the exploit by which the vulnerability is triggered is not pertinent to the evaluation since I am only concerned with the scope of the exploit (in this case, resource exhaustion) rather than its cause. Therefore, the malicious input that caused the exhaustion can be mirrored by attack code that executes to the same effect at some vulnerable code site in the context of the process. Therefore, I generated synthetic attack datasets by separately collecting measurements for exhaustion attack samples and injecting those measurements in the trace of negative (benign) measurements.

The attack measurements were injected once per trace file at a randomly selected location. To account for differences in the total amount of the attacked resource at the injection point, I adjusted
the injected measurements by adding (summing) the last benign measurement vector of the same resource type to each attack vector in the rest of the trace. Being able to inject the attacks at any point in the trace allowed accurate capturing of attacks seen in the wild, and covered even more sophisticated cases, including exhaustion attacks at very early or very late stages in the execution of the process. For example, exhaustion may be possible through attacker controlled environment variables that are used by dynamic libraries during process creation or termination.

The attack snippets were designed to enable the attacks to execute stealthily (by slowly harvesting resources) and avoid early detection. This is a worst-case scenario that is much more conservative than current attacks seen in the wild. For attacks that targeted memory, file descriptors, and tasks, I allocated 10 memory pages, one file descriptor, and one task per each iteration of the attack, respectively. For stack attacks, I used uncontrolled recursion where each stack frame is approximately 512 bytes. CPU attacks were infinite loops that compute \( \sqrt{x} \) and \( x^y \) operations, where each iteration consumed four clock ticks on average. In general, the attacks covered the following CWE classes\(^3\): 400, 401, 404, 674, 770, 771, 772, 773, 774, 775, and 834. Note that the choice of parameters does not bias the results because they do not, by themselves, alter the outcome of the attack or the pattern at which it occurs.

3.4.3 Synthetic Attacks on Apache and W3m

The first experiment replayed a dataset of \( \sim 60K \) unique benign URLs of incoming HTTP GET requests to GMU’s servers. The W3m browser was used on the xterm terminal, and the host domains were mirrored and served using Apache. (On xterm, W3m renders tables, frames, colors, links, and images.) Radmin monitored both Apache and W3m. In the case of Apache, the monitoring was performed per each request handler.

Table 3.3 shows the results for this experiment. Radmin achieved a FPR of only 11 out of 10,000 requests in the case of W3m. For Apache, the number further decreases to only four out of 10,000 requests. In the case of Apache, Radmin saved more than 85\% of the file descriptors (the maximum of negative runs was ten file descriptors). The memory saving for Apache is only five percent due to the highly centralized distribution of memory consumption of Apache during negative runs (1.19GB)

\(^3\)For details and code samples, please refer to the CWE project at [http://cwe.mitre.org](http://cwe.mitre.org)
Table 3.3: Detection performance on Apache and W3m.

<table>
<thead>
<tr>
<th>Prog.</th>
<th>TP</th>
<th>FP</th>
<th>TN</th>
<th>%FPR</th>
<th>CPU (mean ± std.)</th>
<th>File</th>
<th>Task</th>
<th>Mem.</th>
</tr>
</thead>
<tbody>
<tr>
<td>apache-2.4.7</td>
<td>6064</td>
<td>5</td>
<td>12167</td>
<td>0.04</td>
<td>40 ± 23</td>
<td>85 ± 19</td>
<td>12 ± 10</td>
<td>05 ± 03</td>
</tr>
<tr>
<td>w3m-0.5.3</td>
<td>14245</td>
<td>20</td>
<td>18684</td>
<td>0.11</td>
<td>87 ± 08</td>
<td>49 ± 40</td>
<td>25 ± 23</td>
<td>51 ± 27</td>
</tr>
</tbody>
</table>

mean, 1.22GB median, 1.28GB mode). In the case of W3m, the maximum saving achieved was 87% for CPU time (maximum of benign runs was 56 ticks). Overall, the results show that Radmin can save resources effectively with high accuracy.

3.4.4 Synthetic Attacks on Common Linux Programs

The second experiment used ten common Linux programs. The functionality test packages that shipped with the programs were used to train Radmin. The major difference between this experiment and Section 3.4.3 was the lack of input coverage. In Section 3.4.3, I had sufficient input to build a profile of benign behavior with high confidence. In Section 3.4.4, the functionality tests were few, and some of the consumption subsequences were not significant enough to be learned by the model (see Sections 3.3.2 and 3.4.1), resulting in a higher FPR.

The selected programs cover critical infrastructure services used by desktop and web applications — namely, compression, text processing (pattern matching and comparison), hashing, encryption, and remote downloads. Attacks on compression programs can involve highly recursive compressed files (zip bombs), where decompressing the files results in an uncontrolled consumption of CPU time and file descriptors. Attacks on text processing applications use specially crafted regular expressions or data blocks that result in CPU and memory exhaustion. Hashing and encryption are notorious for CPU and memory exhaustion through specially crafted or erroneous messages. Download managers often suffer from the exhaustion of file descriptors and CPU time.

Table 3.4 shows the results of this experiment. As expected, the FPR was higher than Section 3.4.3. Nevertheless, Radmin achieved a low FPR in most of the cases. For earliness, Radmin achieved large savings for all resources, saving more than 90% of CPU time in most cases. This is mainly due to the high skew of the CPU time (in clock ticks) distribution of those programs (e.g., 374 mean, 120
Table 3.4: Detection performance for common Linux programs.

<table>
<thead>
<tr>
<th>Prog.</th>
<th>TP</th>
<th>FP</th>
<th>TN</th>
<th>%FPR</th>
<th>%Erl. (mean ± std.)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU</td>
<td>File</td>
<td>Task</td>
<td>Mem.</td>
<td></td>
</tr>
<tr>
<td>cmp-3.3</td>
<td>9</td>
<td>0</td>
<td>14</td>
<td>0</td>
<td>98 ± 01 62 ± 32</td>
</tr>
<tr>
<td>cpio-2.11</td>
<td>24</td>
<td>0</td>
<td>17</td>
<td>0</td>
<td>99 ± 01 49 ± 35</td>
</tr>
<tr>
<td>diff-3.3</td>
<td>56</td>
<td>0</td>
<td>109</td>
<td>0</td>
<td>90 ± 01 65 ± 32</td>
</tr>
<tr>
<td>gzip-4.0.1</td>
<td>223</td>
<td>2</td>
<td>389</td>
<td>0.51</td>
<td>81 ± 03 50 ± 29</td>
</tr>
<tr>
<td>gawk-4.0.1</td>
<td>109</td>
<td>2</td>
<td>201</td>
<td>0.99</td>
<td>77 ± 28 53 ± 35</td>
</tr>
<tr>
<td>openssl-1.0.1f</td>
<td>380</td>
<td>0</td>
<td>594</td>
<td>0</td>
<td>94 ± 01 77 ± 25</td>
</tr>
<tr>
<td>rhash-1.3.1</td>
<td>22</td>
<td>1</td>
<td>35</td>
<td>2.78</td>
<td>47 ± 40 62 ± 33</td>
</tr>
<tr>
<td>sed-4.2.2</td>
<td>108</td>
<td>6</td>
<td>194</td>
<td>3.00</td>
<td>70 ± 30 62 ± 33</td>
</tr>
<tr>
<td>tar-1.27.1</td>
<td>480</td>
<td>3</td>
<td>980</td>
<td>0.31</td>
<td>98 ± 02 82 ± 24</td>
</tr>
<tr>
<td>wget-1.5</td>
<td>55</td>
<td>0</td>
<td>79</td>
<td>0</td>
<td>95 ± 01 79 ± 21</td>
</tr>
</tbody>
</table>

median, and 1987 mode for tar). Overall, the results demonstrate the effectiveness of our approach and the feasibility of using functionality tests to train Radmin.

I emphasize that the FPR of Radmin is inversely proportional to input coverage. As higher input coverage is achieved, the PFA models used in Radmin become more complete, and the FPR decreases. I discuss this in Section 3.8.1, along with ways to further increase the earliness of detection.

### 3.4.5 Starvation and Livelock Results

In this experiment, I used common resource starvation samples [23, 31, 85, 118]. Simplified snippets of the test cases are provided in [76]. The test cases spanned the two major resource starvation causes: (1) starvation due to a prolonged holding of resources by other processes, and (2) livelocks due to busy-wait locking.

The first test case, filelock, is a multi-process program that managed exclusive access to resources by holding a lock on an external file. Starvation occurred when a process held the lock for a prolonged time, preventing other processes from making progress. A second test case, twolocks, two threads try to acquire two locks in reversed order and release any acquired locks if the two locks were not both acquired. This is a fundamental livelock case due to unordered busy-wait locking of resources. The third test case in this series is a rare bug in sqlite, where two or more threads concurrently fail to acquire a lock.

I ran each test case a 1000 times, and timed out each run after 20 seconds. Runs that finished before the 20 second deadline were considered negative samples, and runs that did not finish by the time limit were considered positive. Table 3.5 shows the results for this experiment.
Table 3.5: Starvation detection performance.

<table>
<thead>
<tr>
<th>Prog.</th>
<th>TP</th>
<th>FN</th>
<th>FP</th>
<th>TN</th>
<th>%TPR</th>
<th>%FPR</th>
<th>%Erl. (mean ± std.)†</th>
</tr>
</thead>
<tbody>
<tr>
<td>filelock</td>
<td>570</td>
<td>0</td>
<td>0</td>
<td>143</td>
<td>100</td>
<td>0</td>
<td>59 ± 26</td>
</tr>
<tr>
<td>twolocks</td>
<td>705</td>
<td>0</td>
<td>0</td>
<td>98</td>
<td>100</td>
<td>0</td>
<td>93 ± 04</td>
</tr>
<tr>
<td>sqlite</td>
<td>460</td>
<td>0</td>
<td>0</td>
<td>180</td>
<td>100</td>
<td>0</td>
<td>76 ± 13</td>
</tr>
</tbody>
</table>

† Erl. here refers only to percentage of saved CPU clock ticks.

Radmin detected the positive samples with high earliness. For filelock, Radmin saved 59% of the maximum (8 ticks) of negative filelock runs. In the case of twolocks, Radmin saved more than 93% of 12 ticks. For sqlite, Radmin saved 76% of 19 clock ticks. Additionally, Radmin achieved 0 FPs and 0 FNs, indicating that none of the negative samples spent time in a PFA state more than the spread of the codewords corresponding to all outgoing transitions from that state. This means that the negative runs showed a set of similar timing behaviors that were fully learned by the model. Due to the external factors involved, for example internal parameters of the kernel scheduler, further studies are needed in order to reach a conclusive understanding of such behavior. Overall, the results show the promise of Radmin in starvation situations that involve multiple processes and threads.

3.4.6 Runtime Overhead

Figure 3.4 summarizes the the overhead incurred by Radmin in the online phase for the programs used in my experiments and for the UnixBench [35] benchmark. I chose UnixBench because it tests various aspects of the system performance and uses well-understood and consistent benchmarks. Note, Radmin generated no false positives for UnixBench. All experiments were executed on machines running Ubuntu Server 14.04, quad-core 2.83 GHz (base) Intel Xeon X3363 processor and 8 GB of memory. Radmin incurred less than 16% overhead, with mean overhead (geometric) of 3.1%. The runtime overhead is more pronounced in CPU bound programs that were more frequently interrupted by the heartbeat thread. Overall, since Radmin avoids sampling, uses static rewriting, and selectively traces a set of events, the overhead incurred is significantly less than generic dynamic instrumentation and profiling tools (more than 200% runtime increase [162, 179]).
3.5 In-the-Wild Attacks

In this section I show results for two of the most widely spread and publicly available DoS attacks against the Apache server: Apache Killer [6], and Slowloris [30]. Exploits for both attacks are publicly available at [5] and [19], respectively.

Apache Killer [6] is a severe vulnerability affecting Apache that was discovered in 2011. It allows attackers to cause Apache to run out of memory by sending requests to retrieve “overlapping” byte ranges of content (using the HTTP Range header). Such requests result in amplified memory usage of the Apache worker threads. The vulnerability affects Apache versions prior to 2.2.21.

Slowloris [30] is a low-bandwidth DoS attack developed in 2009 that uses very slow HTTP requests to take down Apache servers. The attack sends the HTTP request headers as slowly as possible, without hitting the connection timeout limit of the server. If multiple requests are made in parallel, they can consume the entire server’s connections queue, and the server becomes unable to serve legitimate users. The attack typically manifests itself in an abnormally large number of idle or slow sockets.

Figure 3.4: Runtime overhead incurred by Radmin in the online phase for the programs used in Experiments 1 and 2 (left), and the UnixBench benchmark (right).
3.5.1 Procedure

I trained Radmin on a vulnerable version of Apache (2.2.0) and the benign traffic from Section 3.4.3, using a varying number of clients: 10, 50, 100, 150, and 200. For testing, I performed several experiments using blended benign and attack traffic by interleaving benign and attack requests for 75 and 125 attackers (number of attack connections). For Slowloris, the connection rate was set to 10 connections per second. I configured Apache to serve a maximum of 100 concurrent connections. Each experiment was performed twice, with and without Radmin, and for 15 minutes per run. Both benign and attack traffic were started at the same time at the onset of the experiment.

3.5.2 Results

Radmin detected the attacks within an average of four seconds into the attacks. It detected Apache Killer as a memory exhaustion attack, achieving 24% earliness on average. Figure 3.5 shows the total consumption of Apache under the Apache Killer attack with and without Radmin protection. Without Radmin, Apache consumed approximately seven times as much memory. With Radmin enabled, the memory consumption of Apache was successfully confined to that of benign traffic.

Radmin successfully detected the Slowloris attack as socket starvation with 48% earliness on average. I attribute this to the Slowloris connections, which remained open (and idle) for far longer periods than benign connections, resulting in anomalous patterns of socket deallocations. Figure 3.6 shows the difference in the total number of sockets used by Apache with and without Radmin under the Slowloris attack. I also measured the average time each Apache worker thread spent handling a request. This is depicted in Figure 3.7. The figure shows that under the Slowloris attack and without Radmin protection, each worker spent approximately twice the time needed to serve benign traffic. With Radmin enabled, the CPU time consumed by an Apache worker is confined to that of benign traffic.

The Slowloris attack directly impacts the availability of the server since it fills up the connections queue. To illustrate how Radmin thwarts this, I conducted an experiment using only Slowloris attack traffic (100 connections, 10 connections per second, for 600 seconds), and measured server availability (ability to accept incoming connections) with and without Radmin, as observed by the Slowloris attacker. Note that the number of connections is fixed (equalling the server capacity) since this test did not aim to detect volumetric attacks. Figure 3.8 shows the results.
Figure 3.5: Apache total maximum memory usage without (a) and with (b) Radmin against the Apache Killer attack. The total is counted across all Apache worker threads for the duration of the experiment.
Figure 3.6: Apache total maximum open sockets without (a) and with (b) Radmin against the Slowloris attack. The total is counted across all Apache worker threads for the duration of the experiment.
Figure 3.7: Apache CPU usage per worker process without (a) and with (b) Radmin against the Slowloris attack. Here, CPU usage represents the average time spent holding resources by a worker thread from start, and until 1) a response was sent back to the client (benign traffic) or 2) the connection timed out or reset (attack traffic).
Figure 3.8: Apache server availability without and with Radmin against the Slowloris attack. Radmin reduced the server down time from 98.3% to only 7.83% of the attack duration (600 seconds).
Table 3.6: Apache Killer and Slowloris detection performance.

<table>
<thead>
<tr>
<th>Attack</th>
<th>TP</th>
<th>FN</th>
<th>FP</th>
<th>TN</th>
<th>%TPR</th>
<th>%FPR</th>
<th>%Erl. (mean ± std.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Killer</td>
<td>5997</td>
<td>0</td>
<td>81</td>
<td>129478</td>
<td>100</td>
<td>0.062</td>
<td>24 ± 13</td>
</tr>
<tr>
<td>Slowloris</td>
<td>1588</td>
<td>0</td>
<td>104</td>
<td>136982</td>
<td>100</td>
<td>0.076</td>
<td>48 ± 22</td>
</tr>
</tbody>
</table>

In Figure 3.8a, Apache became completely unavailable once Slowloris exhausted all the 100 concurrent connections that Apache was configured to serve at a maximum (around the 10th second), and remained unavailable for the entire experiment duration. In contrast, when Apache was protected by Radmin (Figure 3.8b), the server was only unavailable for a very short period of 47 seconds, starting at the 10th second and ending at the 57th second when Radmin started terminating the Slowloris connections. This effectively reduced the server down time from 98.3% to only 7.83% of the entire 600 seconds the attack lasted. By the 160th second, all Slowloris connections were terminated by Radmin.

Finally, I summarize the detection performance under both attacks in Table 3.6. Overall, Radmin did not miss any of the Apache Killer requests (≈ 6k requests), achieving a TPR of 100%. It triggered 81 FPs out of ≈ 13k benign requests with a FPR of 0.00062. For Slowloris, Radmin achieved a TPR of 100% and a similarly low FPR of only 0.00076. I emphasize that both attacks are eventually detected by Radmin once they exceeded the resources consumed by benign traffic, hence the TPR of 100%, while earliness quantifies the savings achieved by Radmin.

3.6 URadmin: A User Space Only Solution

Using Radmin as presented requires changes to the OS kernel to enable tracing and loading of the kernel tracer module. I argue that opting for such low-level monitoring achieves high monitoring fidelity with inexpensive performance cost.

In this section, I present a solution that completely operates in user space. I show that a comparable overhead can be achieved at the expense of slightly lower detection accuracy and earliness.

URadmin operates as follows: To begin, it statically parses the input executable and marks assembly sites of interest. A site of interest is defined as a basic block of instructions that results in resource consumption. URadmin then instruments the binary by injecting checkpoints (trampoline functions)
Table 3.7: Checkpoint sites in user space.

<table>
<thead>
<tr>
<th>Target Site</th>
<th>Checkpoint Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>strdup, strndup, regcomp*, mmap, mremap, memalign, memdup, malloc, calloc, valloc, realloc, alloca, pvalloc, aligned_alloc, posix_memalign, free*</td>
<td>Memory</td>
</tr>
<tr>
<td>fopen, fdopen, fnmemopen, freopen, dirfd, open, socket, accept, tmpfile, mktemp, mkstemp, mkostemp, mkstemp, mkostemp, close*, fclose*</td>
<td>File descriptors</td>
</tr>
<tr>
<td>Recursive sites*</td>
<td>Stack</td>
</tr>
<tr>
<td>Sites that manipulates the stack pointer</td>
<td></td>
</tr>
<tr>
<td>Every 500 ms.</td>
<td>CPU</td>
</tr>
<tr>
<td>fork*, vfork*, popen*, exec*, exit*, pthread_create, pthread_exit</td>
<td>Child tasks</td>
</tr>
</tbody>
</table>

* The /proc file system is queried for resource usage summaries upon invocation of the call site.

The checkpoints collect resource consumption measurements based on the resource type consumed by the assembly site (see Table 3.7). Checkpoints can be injected either before or after execution of a specific site. URadmin uses information from relevant parameters and return values to call sites of interest to determine when to trigger a checkpoint.

URadmin queries /proc for additional resource usage summaries in situations where monitored call parameters and return value do not provide useful information, e.g., a call to free does not provide the amount of freed memory. Specifically, URadmin reads the VmSize field of /proc/<pid>/status for virtual memory usage, counts open files from /proc/<pid>/fd, and counts child processes by recursively iterating over the ppid field in /proc/<pid>/state. Table 3.7 summarizes when and what checkpoints are injected.

### 3.6.1 Checkpoint Injection

Algorithm 3.2 describes the instrumentation procedure. Instrumenting the binary is achieved by implementing the checkpoints functionalities into the User Tracer as exported library functions. Then, the User Tracer is injected into the binary and calls to the exported checkpoint functions from the
Algorithm 3.2: Instrument Binary

```
input: Binary file $E$
       Checkpoint sites $C$
       User Tracer library $L$

output: Instrumented binary file $E'$

1. $E' \leftarrow E$
2. calls $\leftarrow \text{BuildCallSnippets}(L, C)$
3. foreach procedure $f \in E$ do
4.   foreach assembly site $a \in f$ do
5.     if $a \in C$ then
6.       $\triangleright$ inject a call at site $a$ to the corresponding checkpoint in $L$
7.       $\text{InjectCall}(\text{calls}(a), a, E')$
8. $e \leftarrow \text{EntryPoint}(E)$
9. $\text{InjectCall}(\text{heartbeat}, e, E')$
10. save($E'$);
```

Tracer are injected at their corresponding checkpoint sites in the binary. URadmin then finds the entry point of the binary, i.e., entry of `main` and injects a call to a function in the User Tracer that spawns the heartbeat thread. The heartbeat thread periodically samples the CPU time consumed by the process. Note that since URadmin operates solely in user space, it is not possible to collect CPU time measurements during context switches. Therefore, I use the process CPU time difference between subsequent checkpoints, as seen in user space, as a measure of consumed CPU time. Finally, the modified binary is written to disk.

### 3.6.2 Detection

The rest of the flow takes place at runtime as explained earlier in the case of Radmin. The injected checkpoints collect and send the measurements to the encoder which translates them over $\Sigma$. The PFA engine learns, updates and executes multiple PFAs over $\Sigma$ and the measurements (one PFA per resource type), where each checkpoint results in a transition in one or more PFAs. The detector uses the prediction probability, as computed by the PFA engine, to decide whether to accept or reject the sequences, as described in Algorithm 3.1.
Table 3.8: Detection performance of URadmin.

<table>
<thead>
<tr>
<th>Prog.</th>
<th>TP</th>
<th>FP</th>
<th>TN</th>
<th>%FPR</th>
<th>%Erl. (mean ± std.)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>CPU</td>
</tr>
<tr>
<td>cmp-3.3</td>
<td>9</td>
<td>0</td>
<td>14</td>
<td>0</td>
<td>93 ± 03</td>
</tr>
<tr>
<td>cpio-2.11</td>
<td>24</td>
<td>0</td>
<td>17</td>
<td>0</td>
<td>90 ± 01</td>
</tr>
<tr>
<td>diff-3.1</td>
<td>56</td>
<td>12</td>
<td>97</td>
<td>11.00</td>
<td>84 ± 02</td>
</tr>
<tr>
<td>gauk-4.0.1</td>
<td>223</td>
<td>9</td>
<td>382</td>
<td>2.30</td>
<td>78 ± 02</td>
</tr>
<tr>
<td>gzip-1.6</td>
<td>109</td>
<td>3</td>
<td>200</td>
<td>1.48</td>
<td>75 ± 18</td>
</tr>
<tr>
<td>openssl-1.0.1f</td>
<td>380</td>
<td>7</td>
<td>587</td>
<td>1.18</td>
<td>87 ± 01</td>
</tr>
<tr>
<td>rhash-1.3.1</td>
<td>22</td>
<td>2</td>
<td>34</td>
<td>5.56</td>
<td>35 ± 11</td>
</tr>
<tr>
<td>sed-4.2.2</td>
<td>108</td>
<td>5</td>
<td>194</td>
<td>2.51</td>
<td>66 ± 23</td>
</tr>
<tr>
<td>tar-1.27.1</td>
<td>480</td>
<td>23</td>
<td>960</td>
<td>2.34</td>
<td>96 ± 01</td>
</tr>
<tr>
<td>wget-1.5</td>
<td>55</td>
<td>18</td>
<td>61</td>
<td>22.78</td>
<td>89 ± 02</td>
</tr>
</tbody>
</table>

3.6.3 Evaluation

Table 3.8 shows the breakdown for the detection performance of URadmin. Noticeably, the FPR achieved by URadmin is higher than that of Radmin. This is due to the lack of sufficient monitoring granularity, which prevented URadmin from constructing models as accurately as Radmin. Since URadmin only monitors explicit sites in the binary, it cannot instantly detect when resources are implicitly consumed, e.g., a call to `printf` may allocate memory to construct the formatted string; `malloc` may request memory pages by internally calling `mmap`. Additionally, some programs, e.g., Wget and Openssl used custom memory allocators that can not be fully covered by URadmin without symbol information (symtab information in the ELF binary) that is often stripped from the binary upon release. Such implicit consumptions are lumped together when URadmin queries the `/proc` file system for resource consumption summaries. This results in more spikes in the measurements, which are more likely to be filtered out during the PFA construction since a subsequence involving a spike will have a low significance probability.

In terms of earliness, URadmin achieved significantly less earliness than Radmin. Again, this is due to the lack of sufficient monitoring granularity, which result in coarser codebooks and PFA states. In other words, every symbol in $\Sigma$ for URadmin corresponds to a wider range of measurements when compared to Radmin. This is more evident in resources that can be consumed in chunks, such as CPU and memory. Earliness in terms of file descriptors was comparable to the results obtained from Radmin, since file descriptors are (de)allocated singly.
Figure 3.9 shows the runtime overhead of URadmin compared to Radmin. The ease of deployment of URadmin comes at the expense of additional runtime overhead, which may be prohibitive in practice. URadmin achieved an 18% mean (geometric) runtime overhead, which is significantly higher than that achieved by Radmin (3.1%). This is due to the additional CPU time required to divert the execution flow at each monitored site to collect and record the measurements, as well as the time spent in reading and processing /proc to collect resource summaries. Given these results, I conclude that Radmin is practically more efficient than URadmin in terms of both detection performance and runtime overhead.

In the following section, I analyze the attacker’s knowledge of Radmin, and develop a metric for the degree of vulnerability to resource exhaustion that a program may exhibit even though it is protected by Radmin.

3.7 Analysis of Attacker’s Knowledge

Attackers can attempt to employ Radmin to learn the PFAs for a target program, crafting input that maximizes the consumption by steering the execution to paths of high resource consumption in the PFA. A high resource consumption path in the PFA can have 1) a low probability, meaning it is very unlikely to be exercised by benign inputs; or 2) high probability, meaning it is exercised by benign
inputs. Since Radmin guarantees that low probability paths are detected, the attackers are forced to use high probability paths in the PFAs. However, in that case, the attack cannot count to a resource exhaustion per se. If the PFA contains a high probability path of high resource consumption, some typical benign input to the program does exercise that path and the subsequences of the path are statistically significant. Therefore, the consumed resources cannot amount to exhaustion, or the input would not have been accepted as benign. In this case, rate limiting techniques can be employed to throttle the rate of requests. Nevertheless, Radmin still limits the potential of the attacks to cause resource exhaustion damage by confining the attacks to high probability paths in the PFAs.

To assess the degree of program vulnerability to resource exhaustion, given that it is confined to its PFAs by Radmin, I developed a metric of two values $p_H$ and $D_{vuln}$ corresponding to the probability $p_H$ of reaching a high consumption state, and the estimated amount $D_{vuln}$ of resources that can be wasted given $p_H$. The assumption is that the more likely it is for a program to reach a high resource consumption state in a PFA, and the higher the consumption of that state, the more vulnerable the program is to exhaustion due to benign inputs that exercise that path. An estimate of the amount of vulnerable resources can be computed as the product of the probability of walking a path in the PFA that leads to a high consumption state and the resources consumed along that path. For example, if $p_H = 0.5$, meaning 50% of the inputs $X$ reach a sensitive state, and the consumption of the sensitive state is 100 MB, then it can be estimated that each input in $X$ could waste $D_{vuln} = 0.5 \times 100 = 50$ MB of memory.

I began by constructing the set $\mathcal{H} = \{ \tau(q^i, s_i) \}$ corresponding to the destination states in the PFA of all transitions where the transition symbol $s_i$ belongs to a high resource consumption codeword, i.e., the consumed resources dimension (3rd dimension in a codeword) of the centers $\mu_i$ of the codeword $s_i$ has a large value. Since our definition of $\mathcal{H}$ is local to each program, and for the metric to be comparable across programs, it has to take into account 1) the probability of reaching a high resource consumption state and 2) the amount of resources consumed along the path to that state.

Let $R_j$ be the event of reaching state $j$ from the root $\phi$, let $R_j^c$ be its complement. $R_j$ consists of a set of sub-events $R_j^{(i)}$, each corresponds to the event of reaching $j$ from $\phi$ through some unique path. The probability of $R_j$ can then be computed as $p(\phi, j)$, where $p(i, j)$ is the total path probability from
state \(i\) to state \(j\), given by:

\[
p(i, j) = \begin{cases} 
1, & i = j \\ 
\sum_{k \neq j} \gamma_{kj} p(i, k), & i \neq j
\end{cases}
\] (3.5)

where \(\gamma_{kj}\) is the transition probability from state \(k\) to \(j\). From the definition of \(\mathcal{H}\) it follows that the probability \(p_H\) of reaching any of the states in \(\mathcal{H}\) is the probability of union over the set \(\{ R_j \mid j \in \mathcal{H} \}\) where exactly one \(R_j\) can occur, i.e., exactly one sensitive state is reached at a time, given by:

\[
p_H = P \left( \bigcup_{j \in \mathcal{H}} R_j \right)
\] (3.6)

\[
= \sum_{j \in \mathcal{H}} P \left( R_j \cap \left( \bigcup_{i \neq j} R_i \right) \right).
\] (3.7)

By including the amount of resources consumed by each path \(R_j^{(i)}\) in \(R_j\), and taking the maximum over all expected consumption amounts of each path, we get \(D_{\text{vuln}}\) as:

\[
D_{\text{vuln}} = \max_{j \in \mathcal{H}} \left\{ \left| R_j \right| \sum_{i=1}^{\left| R_j \right|} \mu_{j,3}^{(i)} p \left( R_j^{(i)} \right) \right\},
\] (3.8)

where \(\mu_{j,3}^{(i)}\) is the value of the 3rd dimension of the centers \(\mu_j^{(i)}\) of the inbound transition symbol into state \(j\), in the path traversed by \(R_j^{(i)}\). \(D_{\text{vuln}}\) yields the maximum expected resource amount that is vulnerable to exhaustion. This is necessary since it is possible for paths to exist between different sensitive states with inverse proportional path probabilities and resource amounts. For example, consider a PFA with two paths \(\phi \rightarrow a\) and \(\phi \rightarrow a \rightarrow b\), with path probabilities \(p_a = 0.5\), \(p_b = 0.4\), and memory consumption amounts \(\mu_a^{(3)} = 1024\) MB and \(\mu_b^{(3)} = 2048\) MB, respectively, where both \(a\) and \(b\) are sensitive states. The expected amount of resources consumed by each path is thus \(0.5 \ast 1024 = 512\) MB, and \(0.4 \ast 2048 = 819\) MB, for \(\phi \rightarrow a\) and \(\phi \rightarrow a \rightarrow b\) respectively. In a worst-case scenario, we can assume that the PFA will always make the transition from \(a\) to \(b\), thus consuming more resources than merely stopping at \(a\). Therefore, by taking the maximum over all subevents, I guarantee that the final estimate of \(D_{\text{vuln}}\) accounts for the worst-case scenario of a random walk on the PFA.
$D_{vuln}$ yields values in the same unit as the measurements, where the higher the value the higher the degree of vulnerability and the higher the expected amount of resources that can be exhausted. The value of $p_H$ should be interpreted in the context of how each program operates. For example, rhash operates on fixed blocks of memory, therefore it is very likely to reach its high resource consumption state in every run, i.e., $p_H \simeq 1$ for memory consumption, even though the estimate consumption in this case, $D_{vuln}$, may be less critical than the consumption of other programs.

Algorithm 3.3 describes how to compute $p_H$ and $D_{vuln}$. It proceeds by first filtering out any outgoing transitions from sensitive states in the PFA, then computing $p_H$ over the resulting PFA (lines 1 – 2). Excluding outgoing transitions from sensitive states guarantees that any path through the PFA can reach only one sensitive state at a time, therefore, satisfying Equation (3.7). The algorithm compute $p_H$ by iterating over the states of the filtered PFA, in topological order, cascading their outgoing transition probabilities to the corresponding destination states of the transitions, then summing up the cascaded probabilities of all states in $\mathcal{H}$ (lines 19 – 33). $D_{vuln}$ is computed by taking the maximum of the separately computed estimates of consumption of each cascaded path to a state in $\mathcal{H}$ (lines 3 – 8). In total, the algorithm takes $O(|\mathcal{H}|(|Q + |\tau||))$ where $|\mathcal{H}|$ is the number of sensitive states, $Q$ is the number of states in the PFA, and $|\tau|$ is the total number of transitions.

Table 3.9 shows the computed $p_H$ and $D_{vuln}$ for all the programs used in Section 3.4.3 and Section 3.4.4. I used the highest 10% codewords in $\Sigma$ as sensitive codewords. The results show that Diff and Sed could waste a considerable amount of CPU ticks. In terms of file descriptors, all the programs exposed at least one file descriptor to exhaustion. Apache and Wget both can exhaust two and one child tasks, respectively, even though the probability of reaching a sensitive state is small. Finally, all the programs had a share of memory exhaustion, with Wget coming at the top of the list. I believe this is due to the fact that Wget buffers streams in blocks of fixed size while downloading. Overall, given the value of $D_{vuln}$ for a program, the operator can decide when to throttle inputs based on the available system resources. I argue that, in general, inputs to any program with $p_H > 0.5$ or $D_{vuln} > \frac{\text{system resources}}{\# \text{ of programs}}$ should be throttled, since such programs pose a high threat.
Algorithm 3.3: Compute Exhaustion Metric

\textbf{input} : PFA $M$
\textbf{Sensitive states} $H$

\textbf{output} : $p_H$: probability of reaching a sensitive state
\hspace*{1em} $D_{vuln}$: Estimated amount of resources vulnerable to exhaustion

\begin{itemize}
  \item $\hat{M} \leftarrow \text{Filter}(M, H)$
  \item $p_H, \_ \leftarrow \text{ComputeMetric}(\hat{M}, H)$
  \item $D_{vuln} \leftarrow 0$
  \item \textbf{foreach sensitive state} $h_i \in H$ \textbf{do}
    \begin{itemize}
      \item $\hat{M} \leftarrow \text{Filter}(M, \{h_i\})$
      \item $\_ \cdot D^{(i)}_{vuln} \leftarrow \text{ComputeMetric}(\hat{M}, \{h_i\})$
      \item $D_{vuln} \leftarrow \max(D_{vuln}, D^{(i)}_{vuln})$
    \end{itemize}
  \item \textbf{return} $p_H, D_{vuln}$
\end{itemize}

\begin{procedure}
\textbf{Procedure} Filter($M, H$)
  \begin{itemize}
    \item $\hat{M} \leftarrow \emptyset$
    \item \textbf{foreach transition} $\tau(q_i, s_i) \in M$ \textbf{do}
      \begin{itemize}
        \item \textbf{if} $q_i \notin H$ \textbf{then}
          \begin{itemize}
            \item $\hat{M} \leftarrow \hat{M} \cup \tau(q_i, s_i)$
          \end{itemize}
      \end{itemize}
  \end{itemize}
\textbf{return} $\hat{M}$;
\end{procedure}

\begin{procedure}
\textbf{Procedure} ComputeMetric($\hat{M}, H$)
  \begin{itemize}
    \item $p_H \leftarrow 0$
    \item $q_0 \leftarrow \text{Topological}(\hat{M}, q_0)$
    \item $p \leftarrow (p_0, p_1, p_2, \ldots, p_{|q|})$
    \item $p_0 \leftarrow 1$
    \item \textbf{foreach state} $q_i \in q$ \textbf{do}
      \begin{itemize}
        \item \textbf{foreach outgoing transition} $\tau(q_i, s_i) \rightarrow q_j$ \textbf{do}
          \begin{itemize}
            \item $p_{\text{path}} \leftarrow p_i \ast \gamma(q_i, s_i)$
            \item $p_j \leftarrow p_j + p_{\text{path}}$
            \item \textbf{if} $q_j \in H$ \textbf{then}
              \begin{itemize}
                \item $p_H \leftarrow p_H + p_{\text{path}}$
                \item $D_{vuln} \leftarrow D_{vuln} + \mu_{i,3} \ast p_{\text{path}}$
              \end{itemize}
          \end{itemize}
      \end{itemize}
  \end{itemize}
\textbf{return} $p_H, D_{vuln}$
\end{procedure}
Table 3.9: Degree of vulnerability $D_{\text{vuln}}$ for programs used in our experiments, with the highest 10% of the codewords marked as sensitive. $p_H = \epsilon$ indicates that $p_H < \gamma_{\text{min}}$. The units for $D_{\text{vuln}}$ are 'clock ticks', 'file descriptors', 'child tasks', and 'KB', for CPU, File, Task, and Mem., respectively.

<table>
<thead>
<tr>
<th>Program</th>
<th>CPU $p_H$</th>
<th>CPU $D_{\text{vuln}}$</th>
<th>File $p_H$</th>
<th>File $D_{\text{vuln}}$</th>
<th>Task $p_H$</th>
<th>Task $D_{\text{vuln}}$</th>
<th>Mem. $p_H$</th>
<th>Mem. $D_{\text{vuln}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>apache-2.4.7</td>
<td>0.060</td>
<td>9</td>
<td>0.018</td>
<td>2</td>
<td>0.008</td>
<td>2</td>
<td>0.003</td>
<td>4300</td>
</tr>
<tr>
<td>w3m-0.5.3</td>
<td>0.635</td>
<td>26</td>
<td>0.372</td>
<td>1</td>
<td>0.003</td>
<td>1</td>
<td>0.500</td>
<td>19924</td>
</tr>
<tr>
<td>cmp-3.3</td>
<td>$\epsilon$</td>
<td>0</td>
<td>0.027</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0.833</td>
<td>6028</td>
</tr>
<tr>
<td>cpio-2.11</td>
<td>$\epsilon$</td>
<td>0</td>
<td>0.106</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0.001</td>
<td>214</td>
</tr>
<tr>
<td>diff-3.3</td>
<td>1</td>
<td>428</td>
<td>0.336</td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>0.028</td>
<td>424</td>
</tr>
<tr>
<td>gawk-4.0.1</td>
<td>$\epsilon$</td>
<td>0</td>
<td>0.471</td>
<td>1</td>
<td>0.001</td>
<td>0</td>
<td>0.229</td>
<td>3496</td>
</tr>
<tr>
<td>gzip-1.6</td>
<td>0.055</td>
<td>1</td>
<td>0.158</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>6088</td>
</tr>
<tr>
<td>openssl-1.0.1f</td>
<td>0.034</td>
<td>6</td>
<td>0.283</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>0.961</td>
<td>6044</td>
</tr>
<tr>
<td>rhash-1.3.1</td>
<td>0.081</td>
<td>2</td>
<td>0.096</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>10924</td>
</tr>
<tr>
<td>sed-4.2.2</td>
<td>1</td>
<td>1408</td>
<td>0.262</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>0.039</td>
<td>428</td>
</tr>
<tr>
<td>tar-1.27.1</td>
<td>$\epsilon$</td>
<td>0</td>
<td>0.018</td>
<td>1</td>
<td>0.001</td>
<td>0</td>
<td>0.013</td>
<td>216</td>
</tr>
<tr>
<td>wget-1.6</td>
<td>0.065</td>
<td>1</td>
<td>0.353</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>0.730</td>
<td>27572</td>
</tr>
</tbody>
</table>

3.8 Discussion and Improvements

3.8.1 Higher Accuracy and Earliness

The PFA model used in Radmin learns only the subsequences that have significant prediction probability (see Section 3.3.2), which means that some benign but rare subsequences may not be learned by the PFA. Such subsequences would be erroneously flagged as attacks (false positives) since they traverse low probability paths in the PFA. Although it is straightforward to force the inclusion of such subsequences in the PFA by adjusting the transition probabilities of their corresponding paths, I decided against doing so to give a realistic view of the efficacy of the system. Nevertheless, FPR is inversely proportional to input coverage in Radmin, i.e., as benign input coverage increases, the number of benign rare subsequences decreases and the PFAs eventually become complete. As with any defense system, it is essential to continuously refine and update the models used by Radmin, which can be facilitated by re-learning or by stream PFA construction algorithms (e.g., [71]).

In my evaluation, I used $\gamma_{\text{min}}$, the minimum prediction probability used in constructing the model, as the detection threshold. This allowed a clear distinction between the sequences that the model learned with significant probability and those that were learned with low probability and therefore resulted in false positives. The value of $\gamma_{\text{min}}$, as well as other PFA parameters, were tuned via CV on discrete, hand-picked, hyperparameters. This is arguably not the best practice since false positives can occur. Radmin supports a variant of this approach that avoids false positives by
adjusting $\gamma_{min}$ after training such that each training sequence is predicted with a non-zero probability. More specifically, the effective minimum prediction probability ($\gamma'_{min}$) of the model is used as the detection threshold instead of $\gamma_{min}$. $\gamma'_{min}$ is computed by re-executing the PFAs over their training sequences and computing the minimum prediction probability over all sequences per resource type. This enables feedback training where the PFA parameters are automatically tuned during training using the elbow method over $\gamma'_{min}$. Additionally, using $\gamma'_{min}$ as the detection threshold guarantees that all false positives are significantly different subsequences that were not observed during training.

Leveraging more information about the target process can allow Radmin to achieve higher earliness. For example, input values and attributes can be associated with paths in the PFAs. The challenges here are reaching a reliable model for representing and matching various input vectors (for example, command line arguments, file IO, environment variables) and succinctly associating the input with paths in the PFAs. Given such a model, the PFAs can be traversed without actually executing the program. That would give the near-optimum earliness, since traversing the PFAs is much more cost effective than running the target program itself. Also, static input filters can be synthesized from the PFAs. These ideas will be explored in more details in future work.

3.8.2 Behavior Confinement

Radmin can be used to confine the behavior of processes to users rather than being employed solely to detect anomalous usage. Depending on user activity, Radmin will decipher unique user behavior patterns, helping to defend against attacks by detecting anomalous, but seemingly valid, consumption of resources. Radmin can be extended to seal off paths of infrequent or undesired resource usage in protected programs by adjusting the conditional distributions in the PFAs. Similarly, Radmin can be used to construct a profile of specific behavioral aspects of target programs, such as sequences of executed events or files accessed. It can also confine the behavior of protocols, a focus of future work.

3.8.3 Accuracy of Recursive Sites Identification

Dyninst ParseAPI uses recursive traversal parsing to construct the CFG and employs heuristics to identify functions that are reached only through indirect flows. The resulting CFG may be incomplete, which might cause the User Tracer to miss some recursive code sites if the recursion is chained using
indirect calls that ParseAPI could not resolve. While I argue that such construct is rare in practice, it can be addressed by dynamically tracing indirect calls using a shadow call stack at the expense of increased runtime overhead. I plan on exploring this option as part of future work.

3.8.4 Exhaustion Through Separate Runs

The current monitoring approach considers consumption that lives within individual processes only. This does not allow for detection of attacks that span multiple runs of some targeted program. For example, if a program creates a new file every time it runs, excessive program runs can exhaust the storage space. Extending Radmin to detect exhaustion through separate runs is possible by including PFAs for persistent resources consumed by separate runs of the program over some time period (e.g., multiple days).

3.9 Summary

This chapter presented Radmin as a novel system for early detection of resource exhaustion and starvation attacks. Unlike existing solutions, Radmin does not use static limits but employs both temporal and spatial resource usage information. Radmin reduces the monitoring overhead by hooking into kernel tracepoints. The Radmin user space library keeps track of stack usage consumed by target processes and provides a heartbeat signal that enables Radmin to detect starvation. My experimentation and evaluation has showed that Radmin can detect resource exhaustion and starvation attacks with high earliness (greater than 50% in most cases), high accuracy (less than 0.00076 FPR on Apache), and low overhead (3.1% on average). Comparing two architectures, one with kernel space monitoring and another with user space only monitoring, I have illustrated that kernel space monitoring achieves much higher accuracy and less overhead (at least 6× improvement). I also have analyzed the attacker knowledge and have suggested methods to throttle user requests based on the probability assignments in the PFAs. I have also introduced methodologies for the implementation of Radmin and highlighted possible limitations and areas for improvements.
Chapter 4
Scalable and Accurate Protection against DoS Attacks

Radmin showed promising results; however, it had a quadratic training time complexity in the training data size that makes it prohibitive to apply to large code bases. Radmin also was tested on stateless traffic rather than live sessions. Moreover, Radmin did not cover network state and I/O which are common targets for attacks at the application layer. Another limitation was that Radmin was heavily dependent on “normal” patterns of pure resource utilization without modeling the rate at which individual resources were acquired or used. As I show in this chapter, lack of taking into consideration when individual resources were allocated can lead to prolonged evasion by slow-rate attacks [29], violating the early detection goals of Radmin.

In this chapter, I present Cogo [80] as a novel Probabilistic Finite Automata (PFA) based system for runtime detection and mitigation of software resource exhaustion DoS attacks. Cogo fully addresses all the aforementioned limitations of Radmin enabling the early detection of real-world attacks in many cases before they are able to affect the service operation or quality of service. Cogo operates in two phases: offline and online. In the offline phase, Cogo monitors the entire resource consumption behavior of the target program — including its network I/O — and builds a PFA model that characterizes its resource behavior over time. Cogo monitors network I/O at the individual socket level and supports monitoring of containerized processes. To reduce the model complexity, I introduce an efficient PFA learning algorithm that operates in linear time. During the online phase, Cogo actively monitors the program and detects any deviations from the trained behaviors attributed to the threads and connections that violate them in the program.

To validate my approach, I build a working prototype implementation of Cogo by extending the code base of Radmin [76] to support low-level network I/O monitoring, monitoring containerized processes, and attaching to running processes.1 I discuss two case studies using real-world attacks.

1By building on Radmin, Cogo inherits other monitoring sensors from Radmin, such as CPU and memory sensors.
and commercial-grade testbeds against The Apache HTTP Server [34] and the VoIP OpenSIPS [21] server. In my experiments, Cogo achieved significant improvement in training time over Radmin, requiring only few minutes instead of days to train and build the models. This is significant since in real-world systems training data are expectantly large in size. In addition to short training time, Cogo achieved a low false positive rate (FPR) (0.019% for Apache, 0.063% for OpenSIPS) using small models (76 MB for Apache, 55 MB for OpenSIPS). Cogo swiftly detected the attacks in less than seven seconds into their execution, resulting in zero downtime in some cases. Its runtime overhead is negligible. It increased the latency by $0.2 \pm 0.3$ ms per request for requests that has ten ms latency, resulting in two to three percent latency increase per session.

To summarize, this chapter makes the following contributions:

- Demonstrates Cogo as a novel system for early detection and mitigation of resource exhaustion DoS attacks against real-word complex Internet services. Cogo extends prior work on Radmin [76] by enabling network stack tracing from the application to the kernel, monitoring containerized processes, and attaching to running processes.

- Presents and discusses a linear time training algorithm that reduces the training and model building time complexity.

- Studies the effectiveness of Cogo using realistic testbeds with real-world attacks on Apache and the VoIP OpenSIPS server. The results demonstrate that Cogo is suitable for large-scale deployment as it is scalable, accurate, has low false positives, and can mitigate real-world attacks.

### 4.1 Assumptions and Threat Model

Cogo focuses on DoS attacks that occur at the application layer such as algorithmic and protocol-specific attacks. Volumetric attacks targeting the communication channels or the network and transport layers, as well as other attack vectors such as code execution and memory exposure are outside the scope of this work. I assume that attackers have full knowledge of the internals of the attacked program and can craft benign-looking inputs that prevent the attacked program from serving legitimate clients (a DoS attack). To protect a program with Cogo, I assume the availability of benign
training inputs that cover the typical desired behavior of the program. Cogo uses kernel tracing, and my prototype implementation supports only Linux and Unix-like operating systems since they power the majority of servers.\textsuperscript{2} However, the approach itself does not place restrictions on the runtime environment and can be ported to other operating systems.\textsuperscript{3} Cogo focuses on detection; complete remediation after attack detection should be implemented by the operator and is outside the scope of this work. Nevertheless, Cogo offers the option to migrate the offending process or session to another server, reduce its resource priority, or terminate it based on a configurable policy. Finally, I assume that attackers can be local or remote, but cannot overwrite system binaries or modify the kernel.

4.2 The Cogo System

Cogo operates in two phases: offline training phase, and online detection phase. In the offline phase, Cogo monitors the behavior of the target program on benign inputs and collects a trace of network I/O measurements. The measurements are sequences of raw data that include the event type (socket open, close, send, receive), the consumption amount of the related resource (number of owned sockets and the traffic rate per socket), and meta data such as the PID, the socket inode number, and timestamps.

Similar to Radmin, the raw resource consumption amounts are encoded (quantized) over a countable finite alphabet $\Sigma$ (a finite set of symbols). $|\Sigma|$ is a tuning parameter less than 16 for a maximum of 16 different consumption levels. Encoding is done by mapping (many-to-few) each raw resource consumption value to one symbol from $\Sigma$.\textsuperscript{4} This is necessary since the PFAs (state machines) only work with a finite set of values.

Cogo constructs multiple PFAs from the measurements, one PFA per resource type. The PFAs capture both the spatial and temporal network I/O patterns in the measurements. In the online phase, Cogo executes the PFAs as shadow state machines along with the target program and raises an alarm if a deviation of the normal behavior is detected. Cogo detects anomalous behavior using the statistical properties of the PFAs — namely, the transition probabilities on the PFA edges. In the following, I discuss how Cogo monitors network I/O and its PFA learning and detection algorithms.

\textsuperscript{2}Market share of operating systems: https://en.wikipedia.org/wiki/Usage_share_of_operating_systems.
\textsuperscript{4}We use “measurements” to refer to encoded measurements in the rest of this chapter.
4.2.1 Network Tracing

Cogo monitors the network activity of the target program by intercepting the traffic and socket events that happen in the context of target processes inside the kernel. Specifically, it monitors all socket creation and destruction events triggered by the target processes, and tracks traffic sent or received on those sockets. From the size and direction of the monitored traffic, Cogo computes the transmit (TX) and receive (RX) rates per second.

Cogo differentiates sockets from regular file descriptors inside the kernel as follows: First, it retrieves a target process task structure in kernel space using the global process identifier (PID). (The task structure is the actual structure that represents the process inside the kernel.) It traverses the task structure and extracts the file descriptors table owned by the process. For each file descriptor, Cogo extracts the inode object associated with the file descriptor. (The inode object is a kernel structure that contains all needed information to manipulate and interact with a file descriptor. An inode represents each file in a file system, including regular files and directories, as well as special files such as sockets, devices, and pipes.) Cogo checks if the inode object contains an embedded (allocated member) socket object; if found, Cogo marks the corresponding file descriptor of the inode as a socket descriptor. Cogo tracks all identified sockets by their low-level unique inode numbers throughout their lifetime.

For each identified socket, Cogo extracts the socket Internet protocol family from the socket kernel structure. (The protocol family defines the collection of protocols operating above the Internet Protocol (IP) layer that utilize an IP address format. It can be one of two values: INET6 and INET for the IPv6 and IPv4 protocol families, respectively.) This is essential for determining how to interpret the socket network addresses. Given a socket protocol family, Cogo extracts the local and foreign addresses and port numbers if available. Foreign port numbers may not be available if the socket is a listening or a datagram socket.

Cogo intercepts all transmit and receive socket events that occur in the context of the monitored process in kernel space. This includes regular I/O operations such as streamed and datagram I/O, in addition to asynchronous I/O (AIO) operations and operations utilizing a socket iterator. Cogo collects the direction (TX or RX) and size of the traffic, and associates that with the corresponding socket inode number. The TX and RX rates are computed periodically per socket. The period length is 53
configurable and defaults to one second. To minimize memory and runtime overhead, Cogo installs a kernel timer that ticks once per period length. This requires minimal memory per socket as only the last tick timestamp and total traffic size need be kept in memory. It also minimizes runtime overhead by avoiding unnecessary context switches to compute the rates. Cogo also monitors the socket status, namely connected or disconnected. When a socket disconnects or is unallocated by the kernel, Cogo purges any structures associated with that particular socket from its kernel memory.

### 4.2.2 Training & Learning

Cogo employs Probabilistic Finite Automata (PFA) based learning and detection. Cogo builds one PFA for each monitored resource: one PFA for socket creation and destruction, one PFA for TX rate, and one PFA for RX rate. Cogo uses the PFAs to compute the probability of observed measurements in the online phase. The proposed training algorithm runs in time linear in the measurements length, making Cogo very attractive and realistic for real-world deployment.

#### Constructing Bounded Generalized Suffix Trees

To construct each resource PFA, first, Cogo constructs a bounded Generalized Suffix Tree (GST) from the resource measurements. (A suffix tree is a tree containing all suffixes of a given string. A GST is a suffix tree for a set of strings.) Given a set of strings \( S \) over an alphabet \( \Sigma \) (a finite set of symbols), a GST over \( S \) contains a path from the root to some leaf node for each suffix in \( S \). Each edge in the GST is labeled with a non-empty substring in \( S \); labels of outgoing edges from the same node must begin with unique symbols. A GST can be constructed in linear time and space \( O(n) \), where \( n \) is the total number of symbols in \( S \), using Ukkonen’s algorithm [163]. A GST allows efficient implementations of several string query operations such as linear time substring searching and finding the longest common substring among all the strings in the set. Cogo bounds the depth of the GST by processing the measurements into non-overlapping subsequences of maximum length \( L \). This bounds the depth of the GST to \( L \) and the space requirements per GST to \( O(|S|L) \).

After constructing the bounded GST, Cogo counts the number of occurrences of each substring in the tree. This corresponds to the number of leaf nodes in the subtree rooted at each node in the tree. These counts are computed in a single depth-first traversal of the GST. For each parent-child nodes in
the tree, the ratio between the child’s count to the parent’s count gives the conditional probability of seeing the first symbol of the corresponding child substring after the parent’s. More formally, the prediction probability of a symbol $s_j$ after a substring $s_is_{i+1}...s_{j-1}$ can be computed as:

$$P(s_j|s_is_{i+1}...s_{j-1}) = \frac{\text{count}(s_is_{i+1}...s_{j-1}s_j)}{\text{count}(s_is_{i+1}...s_{j-1})},$$

which Cogo computes on-the-fly during the depth-first traversal of the GST to count the substrings and stores it in each child node in the tree.

**Inferring the PFAs**

Cogo infers a PFA from the GST. Each PFA is a 5-tuple $(\Sigma, Q, \pi, \tau, \gamma)$, where: $\Sigma$ is a finite set of symbols processed by the PFA; $Q$ is a finite set of states, and $q^0 \in Q$ is the start state; $\tau: Q \times \Sigma \to Q$ is the state transition function; and, $\gamma: Q \times \Sigma \to [0, 1]$ is the transition probability function.

To infer a PFA from the GST, Cogo starts by creating a forest of unconnected PFA nodes. Each node has a unique ID and corresponds to exactly one node in the GST. It then traverses the GST in depth-first order: For each edge between each parent (source) and child (destination) nodes in the GST, Cogo checks the length of the edge label. If the edge label has exactly one symbol, Cogo adds a transition between the corresponding source and destination nodes in the PFA. In this case, it sets the transition probability to the child node probability in the GST and sets the transition symbol to the edge label. If the edge has a label of length greater than one, i.e., the label is a substring consisting of multiple symbols, then Cogo adds nodes to the PFA corresponding to each inner symbol in the label; adds a PFA transition from the source state to the node corresponding to the first symbol in the label; and adds another transition from the last inner symbol in the label to the destination node.

Formally put, given the edge $u \xrightarrow{s_is_{i+1}...s_{j-1}} v$ in the GST, Cogo adds the following path to the PFA:

$$u' \stackrel{s_i, \text{count}(u[s_i]) / \text{count}(u)}{\rightarrow} \bullet \stackrel{s_{i+1}, 1.0}{\rightarrow} \ldots \stackrel{s_{j-1}, 1.0}{\rightarrow} \bullet \stackrel{s_j, 1.0}{\rightarrow} v',$$

where $u'$ and $v'$ are the corresponding nodes in the PFA of $u$ and $v$. Recall that transitions in the PFA hold both a transition symbol and an emitted probability.

This initial PFA contains paths that correspond to the substrings from the GST and can be used for prediction so long as the *entire* substring is in the tree. However, if the next symbol following
some substring is not in the tree, a Markovian decision need be made since it may still be possible to predict the symbol using a shorter suffix. For this, the GST suffix links are used to find the next immediate suffix. In a GST, the node corresponding to the string \( s_i \ldots s_j \) has a suffix link (a pointer) to the internal node corresponding to the string \( s_{i+1} \ldots s_j \), i.e., its immediate suffix. This enables jumping to the next available context (sequence history) in constant time. Cogo utilizes the suffix links to complete the PFA construction in the following manner: For each node \( u \) (visited during the depth-first traversal) and for each symbol \( \sigma \) that does not mark any outgoing edge from \( u \), Cogo follows the suffix links starting from \( u \) until:

1. An internal node \( v \) is reached where the first symbol of the substring represented by that node equals \( \sigma \). In this case, Cogo adds a transition between the corresponding PFA nodes to \( u \) and \( v \), sets the the transition symbol to \( \sigma \), and the transition probability to that stored in \( v \) in the GST.

2. The root of the GST is reached, and it has an edge with a label that begins with \( \sigma \) to some child node \( v \). Here, Cogo adds a transition between the corresponding \( u \) and \( v \) nodes in the PFA. It sets the transition symbol to \( \sigma \), and the transition probability to that stored in \( v \).

3. The root is reached, but it has no outgoing edges for \( \sigma \). In this case, a loop-back transition on \( \sigma \) from \( u \) to itself is added, and the transition probability is set to \( \rho_{\text{min}} \) (a small predefined value for the minimum transition probability).

Since the GST contains all suffixes, the resulting PFA contains outgoing edges from the start state that never prefixed the training sequences. This can result in the PFA accepting anomalous behavior if an attack occurs at the execution onset of a target process. Cogo eliminates these spurious transitions in constant time \((|\Sigma|\) comparisons) by keeping a set of the initials of the training sequences and pruning outgoing start state transitions from the PFA that do not correspond to those initials. Using a single depth-first traversal, Cogo also removes any transitions that have a probability less than or equal to the minimum transition probability \( \rho_{\text{min}} \). It replaces these transitions with loop-back transitions with \( \rho_{\text{min}} \) probability. Finally, during the same traversal, Cogo normalizes the probabilities across outgoing edges from each node.
Minimizing the PFAs

The PFA may contain redundancy such as unreachable states (because of eliminated transitions) or overlapping paths, resulting in unnecessary space overhead. To overcome this, Cogo attempts to reduce the size of the PFA as much as possible, but without incurring excessive training overhead, i.e., reduction time has to be linear in the size of the PFA. This minimization algorithm is based on the insight that paths farther away from the PFA root (the start state) are more likely to overlap since they represent longer substrings.

Cogo iterates over the PFA in breadth-first order; each time it visits a new state $u$, it searches for all previously visited states that are fully equivalent to $u$. In a PFA, two states are fully equivalent if they have the same outgoing transitions with the same transition symbols, probabilities and destination states. Cogo groups all the equivalent states into a single state set. This process continues till all states in the PFA are visited, producing a set family of states. Then, all equivalent states set are removed and replaced with a single state in the PFA. The process is repeated on the resultant PFA till any of the following conditions occur: 1) The PFA stops changing. 2) The minimization ratio, i.e., the size of the resulting PFA divided by the size of the old PFA, drops below some user chosen threshold $\theta$ (default to 0.1). 3) The number of repetitions exceeds a user chosen threshold $\zeta$ (default to 100). The 2nd condition terminates the minimization stage once a diminishing returns point is reached. The 3rd condition gives the user the ability to control the hidden constant $c$ of the minimization complexity (at most $O(cn)$). This completes the construction of the PFA. Figure 4.1 shows an example of a bounded GST and the PFA inferred by Cogo from the set $\{01001101, 01010100\}$, where $L = 4$, i.e., the effective set is $\{0100, 1101, 0101, 0100\}$. The figure illustrates how to compute the probability of the sequence 010 using the PFA.

4.2.3 Detection

In the online phase, Cogo executes the PFAs as shadow state machines to the monitored program. Each measurement symbol results in a transition in the corresponding PFA of that measured resource type. As explained in Chapter 3, computing the probability of a sequence of symbols using a PFA reduces to walking the path corresponding to the symbols in the PFA, one transition at a time. This
enables constant time online detection with minimal state keeping overhead since only the current state and the transition symbol determine the next state.

*Cogo* decides that the sequence *s* is anomalous if the sequence results in at least *t* low probability transition in the PFA. Specifically, *Cogo* performs the following test:

\[
\left\{ \gamma(q_j, s_i) \leq \rho_{\text{min}}, i \in 1 \ldots t \right\} \leq t \rightarrow \text{accept} \bigg\vert \bigg\{ t \rightarrow \text{reject} \right. \quad (4.2)
\]

where \(\gamma(q_j, s_i)\) is the transition probability of symbol *s*\(_i\) outgoing from state *q*\(_j\), \(q_{j+1} = \tau(q_j, s_i)\) gives the next PFA state, and *t* is the tolerance level. Recall that *Cogo* builds the PFAs such that low probability transitions are loop-back transitions, therefore they do not result in a state change in the PFA. This allows *Cogo* to offer tolerance by forgetting up to *t* low probability transitions; *Cogo* raises an alarm if a sequence results in more than *t* low probability transitions.

### 4.2.4 Attaching to a Running Process

It is practically desirable to be able to attach *Cogo* to an already running process. For instance, this facilitates on-demand monitoring of processes that migrate among a cluster of servers. The main challenge in attaching to a running process is that *Cogo* would not know in which states in the PFAs the process might be nor how it got to those states. The process and the PFAs would not be in sync.
I developed the following non-deterministic PFA executor to support attaching to a running process: First, Cogo attaches to the running program and starts monitoring at any arbitrary point in its execution. As measurements arrive, and for each PFA for the target program, Cogo executes the PFA in a non-deterministic fashion by finding all paths that correspond to the incoming measurements. This gives a set of potential paths $P$ that the monitored process might have executed along. Next, as more measurements arrive, Cogo extends each path in $P$, one transition at a time, and checks if the detector accepts or rejects the new paths. A rejected path is eliminated from $P$. Eventually, either all paths in $P$ are eliminated or only a single path remains. If all paths are eliminated, meaning the process has deviated, Cogo raises an alarm. If a single path remains, then the PFA and the process have been successfully synchronized and Cogo returns to normal operation.

### 4.2.5 Seeing Through Containers

Web applications are typically deployed in isolated instances. Multiple instances of the web server would be running in isolation from each other on the same host machine. Each instance gets its own isolated view of the host resources — including file system, CPU, RAM, and network interfaces. Common isolation techniques are based on full virtualization (e.g., virtual machines) or operating-system-level virtualization using software containers (e.g., OpenVZ, LXC, Docker). Full virtualization does not pose an issue for Cogo since Cogo can be deployed inside the web server virtual machine. On the other hand, containers abstract out the OS kernel, preventing the deployment of Cogo inside an isolated container since Cogo requires kernel access. Therefore, Cogo has to be deployed on the host OS (outside the containers) yet monitor processes running in isolated containers.

The main hurdle of monitoring containers is that PIDs inside a container are local to that container, i.e., they only identify the process inside that container PID namespace. Quoting from the Linux kernel manual, “a namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource.” The local PID serves no meaning outside the container in which the process is running. Instead, the process is identified by a different global PID only known to the host running the container. Cogo cannot attach and monitor a process in kernel space without knowledge of its global PID since the

---

process is identified in kernel space by its global PID. (There are no containers or namespaces in kernel space.)

I implemented a container-aware global PID resolver to support monitoring containerized processes. It operates as follows: First, Cogo starts the process in a suspended state inside the container and gets the process ID in the container namespace (NSPID). This is achieved by using a custom loader process that outputs its NSPID and its namespace identifier (NSID) then sends a stop signal to itself. (The NSID is a unique namespace identifier.) When the loader process receives a continue signal, it loads the desired target program via a call to the exec system call. The NSPID reported by the loader process is the PID local to the container where the loader is running (and eventually the target program). Given the NSPID and NSID, Cogo then searches all namespaces on the host system for a matching child NSID that contains a process with a matching NSPID. Once identified, Cogo extracts the global PID of the process, i.e., the actual PID on the host system corresponding to the containerized process NSPID. It then attaches to that process (the loader) using the global PID and sends it a continue signal. Upon receiving the continue signal by the loader, it loads and executes the desired target using the exec system call, replacing the process image but retaining any PIDs. Cogo then continues normal operation.

4.3 Implementation

I implemented Cogo by extending the code base of Radmin [76]. Figure 4.2 illustrates the architecture of Cogo within Radmin. I extended Radmin’s kernel tracer to support network I/O monitoring, and implemented Cogo’s learning and detection algorithms by extending Radmin’s PFA engine. I also extended the framework to support attaching to running processes and monitoring containerized processes.

I extended Radmin’s kernel tracer to support network I/O monitoring by attaching handlers to the relevant tracepoints [73] in the kernel. Kernel tracepoints are special points in the executable kernel memory that provide hooks to various events in the kernel. The hooks call functions (probes) that are provided at runtime by kernel modules. Cogo provided a handler for each tracepoint, where it collected and reported the measurements to the rest of Radmin as needed. Each tracepoint executes
Figure 4.2: Cogo’s architecture within Radmin. Cogo extends Radmin with a network I/O monitoring module, the linear PFA construction component (l-PFA in the figure), a non-deterministic PFA executor, and a custom loader to resolve namespace PIDs.

Table 4.1: Kernel tracepoints hooked by Cogo for network I/O monitoring.

<table>
<thead>
<tr>
<th>Kernel Tracepoint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>socket.create</td>
<td>A socket is allocated</td>
</tr>
<tr>
<td>socket.close</td>
<td>A socket is closed and released</td>
</tr>
<tr>
<td>socket.sendmsg, socket.writev, socket.aoi_write, socket.write_iter</td>
<td>Data is being sent on a socket</td>
</tr>
<tr>
<td>socket.recvmsg, socket.readev, socket.aoi_read, socket.read_iter</td>
<td>Data is received on a socket</td>
</tr>
</tbody>
</table>

in the context of the process that triggered the event. Cogo filters out process contexts using the PID of the monitored process. It supports monitoring a single process or all processes in a process tree.

Table 4.1 lists the relevant tracepoints hooked by Cogo to monitor network state.

4.4 Evaluation

I measured the detection accuracy, earliness and overhead of Cogo on two large-scale server applications commonly targeted by application layer DoS attacks: Apache [34], the world’s most used web server software; and OpenSIPS [21], the famous free VoIP server and proxy implementation of the session initiation protocol (SIP) [140]. The testbeds used Docker containers for isolation and used CORE [42] for network emulation.
4.4.1 HTTP Attacks on Apache

The Apache testbed is depicted in Figure 4.3. The testbed consisted of a server running Apache, one User Agent (UA) node for benign clients, and one Weaponized User Agent (W-UA) node for attackers. Each of the UA and W-UA nodes consisted of a Docker container running HTTP clients. We generated benign traffic using an HTTP client model derived from the Choi-Limb model [64]. From the mean and standard deviation for various model parameters for the data set reported in [64], a nonlinear solver was used to calculate approximate distribution parameters for the distribution found to be a good fit in Choi-Limb. Each client was represented using one instance of the HTTPerf [13] benchmark. For each client, a workload session was generated using a unique seed and the distilled distribution parameters. The session consisted of a series of requests with a variable think time between requests drawn from the client model. The session was generated in HTTPerf’s workload session log format (wsesslog). Each client request contained as URL parameters a random request length padding and a requested response size drawn from the client model. The Apache server hosted a CGI-bin web application that simulated real deployments. For each client HTTP request, the server responded with a message body with content equal in byte length to the requested response size.

Attack traffic originated from the W-UA node, using the HTTP application layer DoS benchmark SlowHTTPTest [7]. SlowHTTPTest bundles several Slow-Rate [29] attack: low-bandwidth application layer DoS attacks that use legitimate albeit slow HTTP requests to take down web servers. Two famous examples of slow-rate attacks are Slowloris [30] and Slowread [8]. In Slowloris, attackers
send the HTTP request headers as slowly as possible without hitting the connection timeout limit of the server. Its Slowread variant sends the headers at normal speeds but reads the response as slowly as possible. If enough such slow requests are made in parallel, they can consume the entire server’s application layer connections queue and the server becomes unable to serve legitimate users. Slow-Rate attacks typically manifest in an abnormally large number of relatively idle or slow sockets.

The Cogo’s models for Apache were built using 12 benign traffic runs, each of which consisted of one hour of benign traffic. The number of benign clients was set to 100. Each benign client was a whole workload session. For testing, I performed several experiments using blended benign and attack traffic by injecting attack requests at random points while serving a benign load. Testing was performed by running Apache under Cogo in detection mode, and serving one hour worth of benign requests from 100 benign clients and 100 slow-rate clients (attackers). The number of attackers represents the total concurrent SlowHTTPTest attack connections. The attack duration was limited to 15 minutes; Apache was configured to serve a maximum of 100 concurrent connections at any moment in time.

I performed each experiment twice, with and without Cogo. I configured Cogo to kill the offending Apache worker process when an attack is detected in its context. Finally, I experimented with two types of attackers: non-aggressive attackers that seep in the server at a very slow rate, and aggressive attackers that bombard the server with as many concurrent connections as possible. For non-aggressive attackers, the SlowHTTPTest connection rate was set to one connection per second; for aggressive attackers, it was set to the server capacity, i.e., 100 connections per second.

Detection Results

Table 4.2 summarizes the results. It took Cogo only about 12 minutes to build a model from the benign measurements. This is about a $505 \times$ improvement over Radmin which took more than 4 days to construct a model from the same measurements. The savings in training time came at the expense of a slight increase in the model size (from 34 MB to 76 MB) which is acceptable and does not pose a bottleneck. The model is only loaded once at startup of Cogo; detection time is invariant of the model size as each measurement point results in exactly one transition in one of the PFAs.
Table 4.2: Summary of results for Apache. The number of requests was 473,558.

<table>
<thead>
<tr>
<th>Item</th>
<th>Radmin</th>
<th>Cogo</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training time (sec.)</td>
<td>379,661</td>
<td>752</td>
<td>▼505×</td>
</tr>
<tr>
<td>Model size (MB)</td>
<td>34</td>
<td>76</td>
<td>▲0.45×</td>
</tr>
<tr>
<td>FPs, FPR</td>
<td>1,116, 0.2357%</td>
<td>92, 0.0194%</td>
<td>▼12×</td>
</tr>
<tr>
<td>Downtime (sec; non-aggressive)</td>
<td>137</td>
<td>0</td>
<td>▼∞</td>
</tr>
<tr>
<td>Downtime (sec; aggressive)</td>
<td>58</td>
<td>7</td>
<td>▼8.3×</td>
</tr>
</tbody>
</table>

Cogo achieved a low false positive rate (FPR) at 0.0194% (about 91% better than Radmin). I believe the reason for this reduction in FPR is that Cogo retains longer low-probability paths in the PFA as the detection algorithm limits transition probabilities rather than whole path probabilities as in Radmin. For the most part, false positives (FPs) were encountered during startup or shutdown of Apache, which from experience has shown considerable variability.

Figures 4.4 and 4.5 depict the availability of Apache against non-aggressive and aggressive attacks. Cogo successfully prevented Apache from going down against non-aggressive attacks. As the attack connections were idling at the server side, Cogo detected anomalous transmit and receive rates and terminated the attacked Apache workers. This occurred within seven seconds from connection establishment. Against the same attacks, Apache under Radmin remained down for longer than two minutes. For aggressive attacks, Apache protected with Radmin was down for one minute, compared to only only seven seconds under Cogo.
Figure 4.4: Apache server availability against non-aggressive slow-rate attacks. With Radmin, the server was down for more than two minutes. There was no downtime under Cogo.
Figure 4.5: Apache server availability against aggressive slow-rate attacks. Cogo reduced the server down time by at least a factor of eight, down from 58 seconds to only 7 seconds.
4.4.2 VoIP Attacks on OpenSIPS

Next, I considered detection of resource attacks on VoIP servers as telephony systems have increasingly become targets of DDoS attacks evidenced during the 2015 attack on the Ukrainian power grid [38]. To establish and manage calls, VoIP servers rely on Session Initiation Protocol (SIP) [140] which is known to be vulnerable to exhaustion and overload, even under benign conditions [93]. Overload can be caused by a variety of legitimate SIP behaviors such as response duplication, call forwarding, and call forking (conference calls) which result in large numbers of control packets that may congest servers. Similarly, excessive transactions cause system resource exhaustion in stateful servers when the number of requests exceeds the finite memory available to track each call state machine. An adversary who wishes to cause DoS can do so by initiating calls that exercise these legitimate but atypical resource intensive behaviors and thus degrade server performance — all while blending in with normal traffic (without malformed packets or specification violations) to circumvent defenses such as scrubbing or bandwidth limitation. In the following, I evaluate Cogo against these protocol attacks on a representative SIP testbed based on OpenSIPS [21].

Testbed and Procedure

The SIP DDoS testbed, shown in Figure 4.6, consisted of a SIP server and pairs of SIP user agents and weaponized agents that serviced simultaneous callers and attackers. The SIP server ran OpenSIPS 2.2 and was configured using the residential configuration generated by the OpenSIPS configuration tools. OpenSIPS used fixed-size shared and private memory across its child processes (32 MB and 16 MB respectively). To exacerbate memory exhaustion at the server, the wt_timer of OpenSIPS was adjusted to 32 seconds (the recommended value in the RFC) which corresponds to the length of time a transaction is held in memory after it has completed. Though intended to help absorb delayed messages after the transaction completed, it also inadvertently reserves memory that could otherwise be made available to handle new calls. For the following experiments, I considered a small enterprise or large residential deployment, thus end-to-end delays from UAs to the server were minimal (about ten ms) and link bandwidth was isolated to SIP traffic at 100 Mbps.

Pairs of UA nodes were used to represent benign SIP callers (UA-1) and callees (UA-2). These nodes ran instances of the SIP Proxy (SIPp) [28]: a SIP benchmarking tool to generate SIP caller/callee
workloads. While the audio portion of the call was not modeled, the log-normal feature of SIPp was leveraged to insert a random, lognormal distributed pause between call setup and hang up to simulate variability among call lengths. The call length distribution was log-normal with a mean of 10.28 and variance of 1 ms equating to an average call length of 30 seconds. Each call consisted of an INVITE transaction followed by the variable pause, then terminated with a BYE transaction. SIPp can initiate calls in parallel, allowing modeling of many users from a single node.

Attacks were initiated from the W-UAs at caller W-UA-1 and callee W-UA-2. They were staged by repurposing SIPp as an attack tool, supplying it with scenario files that specify malicious caller/callee behaviors such as flooding requests, or excessive duplication of responses. For example, a BYE flood attack equates to W-UA-1 initiating a number of spurious BYE transactions, each with a new call id to represent a new transaction. Because SIP does not associate a BYE with a prior INVITE, the BYE is accepted and transaction memory is wastefully reserved while the attack is in process. W-UA-2 colludes with W-UA-1 by purposefully not responding to the request, which adds to the time transaction memory is held at the server. Like the benign workload, the amplitude of SIPp can be tuned to control the number of simultaneous attack calls.

The Cogo model for OpenSIPS was built from five benign observation collecting runs, totaling eight hours of benign measurements. During this observation run OpenSIPS was subjected to a benign load between the SIPp clients (UA-1, UA-2) and the SIP server. The clients initiated calls to the server. Call setup and call disconnect were specified using XML files input to SIPp and followed standard SIP call setup conventions for invite, ringing, bye, and appropriate response and status
Table 4.3: Summary of OpenSIPS results. The number of benign call requests was 6,342.

<table>
<thead>
<tr>
<th>Item</th>
<th>Radmin</th>
<th>Cogo</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training time (sec.)</td>
<td>43,493</td>
<td>258</td>
<td>▼169×</td>
</tr>
<tr>
<td>Model size (MB)</td>
<td>41</td>
<td>55</td>
<td>▲0.75×</td>
</tr>
<tr>
<td>FPs, FPR</td>
<td>9, 0.1419%</td>
<td>4, 0.0631%</td>
<td>▼2.25×</td>
</tr>
<tr>
<td>Bye flood Detection delay (sec.)</td>
<td>∞</td>
<td>6</td>
<td>▼∞</td>
</tr>
<tr>
<td>Invite flood Detection delay (sec.)</td>
<td>∞</td>
<td>4</td>
<td>▼∞</td>
</tr>
</tbody>
</table>

messages. Call hold used the SIPp log-normal distribution. The SIPp maximum calls per second rate was set to 10 and call limit to 200. This combination of SIPp settings produced a steady call rate of ~7 calls every second. Several additional benign observation runs were made during which OpenSIPS was started and then terminated to ensure the observations captured startup and shutdown which from experience has shown considerable variability. The total size of the observation data was 515 MB. Processing these observations resulted in a model of 55 MB. Several test runs were made using the model and with it Cogo exhibited virtually zero false positives under a load with the same characteristics as that used for the observation runs.

Detection Results

Table 4.3 summarizes the results. Cogo reduced training time from about 12 hours to only four minutes (greater than 169× reduction). The model size increased by a factor of only 0.75 (from 41 MB to 55 MB). In terms of accuracy, Cogo only had four FPs throughout the experiment, all occurred at startup time of OpenSIPS. Radmin triggered nine FPs also at startup time. The impact of BYE and INVITE floods on OpenSIPS, and the detection behavior of Cogo is shown in Figure 4.7. The attacks were not detectable by Radmin since OpenSIPS uses a fixed-size memory pool, therefore preventing exhaustion of memory by attack calls. Without monitoring network I/O, it is impossible to detect BYE and INVITE floods in their early stages. Cogo, on the other hand, detected the attacks within less than six seconds into the attacks. Note that I did not implement a remediation policy for OpenSIPS since proper remediation requires a protocol-specific solution that times out or hangs up malicious calls.
4.4.3 Performance Overhead

Cogo effectively had a negligible overhead. I measured the throughput of Apache and OpenSIPS on the benign workloads with and without Cogo. Apache maintained a steady rate of 130 requests per second. Then, I benchmarked Apache with HTTPerf and experienced a very marginal 0 ± 0 ms response time increase per request. The average response time increased from 10 ms to 10.5 ms. For OpenSIPS, it maintained a steady call rate of 200 calls per second. Experiments were conducted with call rates from 300 to 1000 calls per second and no degradation in throughput were observed.

4.5 Summary

This chapter presented Cogo, a practical and accurate system for early detection of DoS attacks at the application layer. Unlike prior solutions, Cogo builds a PFA model from the temporal and spatial resource usage information in linear time. Cogo monitors network state, and supports containerized processes monitoring and attaching to running processes. Cogo detected real-world attacks on Apache and OpenSIPS, both are large-scale servers. It achieved high accuracy, early detection, and incurred negligible overhead. Cogo required less than 12 minutes of training time, incurred less than 0.0194% false positives rate, detected a wide range of attacks less than 7 seconds into the attacks, and had a negligible response time overhead of only 0.2 ± 0.3 ms. Cogo is both scalable and accurate, suitable for large-scale deployment.
Since its introduction by Shacham in 2007 [152], Return-Oriented Programming (ROP) has become an increasingly popular technique for bypassing Data Execution Prevention (DEP) defenses on modern operating systems. DEP mitigates classical code injection attacks by ensuring that all writable memory pages of a program are non-executable, preventing the execution of any input data. ROP attacks counter DEP by executing arbitrary functionalities without injecting new code. Instead, existing sequences of instructions in the process executable memory (called gadgets) are chained together to perform the intended computation. While Address Space Layout Randomization (ASLR) [3] randomizes the location of most libraries and executables, ROP attacks can still bypass ASLR by finding a few code segments in statically known locations, or through brute-forcing and de-randomization by exploiting memory disclosure vulnerabilities [153, 156].

5.1 Return-Oriented Programming (ROP)

Return-Oriented Programming (ROP) [152] is a generalization of return-into-libc attacks [132]. Rather than causing the program to return to known libc functions, ROP returns to arbitrary instruction sequences in the executable memory of the program, enabling attackers to execute arbitrary code without injecting any new code in the process memory.

The basic idea behind ROP is to use indirect jumps (e.g., `ret` instructions) to return to arbitrary points in the executable process memory that execute sequences of instructions ending in another indirect jump instruction.\(^1\) The last indirect jump instruction allows executing one instructions sequence after another. Multiple sequences can be combined into “gadgets” that perform an atomic

\(^1\)Execution can return to either indented (i.e., beginning of instructions) or unindented points (i.e., any byte) in the executable process memory.
task (e.g., load, store, system call). The attacker then chains the gadgets together to perform the intended malicious functionality. Typically, gadgets end with a ret instruction which returns to the stack. The attacker chains the gadgets by hijacking the stack and writing appropriate addresses to the beginning of the desired instruction sequence.

A typical ROP attack operates as follows: First, the attacker overwrites the stack contents with addresses of the desired ROP gadgets. Once the ret instruction of the current routine is executed, the first return address of the current stack frame is used as a return target. Instruction sequences at that address will execute till the next ret instruction. Control is transferred to the next gadget upon execution of the ret instruction. This process repeats, jumping from one gadget to the next, till the gadget chain terminates. Figure 5.1 illustrates a gadget that stores a constant 0x1 at the target memory address 0xa0de1b6e. The gadget starts by loading the constant 0x1 from the stack to the register eax. It then loads the target memory address to register ebx. Finally, it moves the contents of eax back to the memory pointed at by ebx.

Conventional ROP attacks use ret instructions to chain the gadgets [53, 60, 97, 108, 138, 152]. In [50], a ROP variant was presented that uses indirect jump (e.g., jmp eax) instructions to chain
the gadgets. It has been shown that ROP can perform Turing-Complete computations if the attacker can find sufficient gadgets to perform all of the following: memory, arithmetic, logical operations, and system calls [53, 137, 161]. A famous example is the recent ROP-only Adobe Reader exploit [4]. I refer the reader to [134, 152] for more details on ROP.

It is worth mentioning that overwriting the return address on the stack is not the only way to hijack the execution of a target process. Other vulnerabilities — including format string, memory corruption, and integer overflow — can allow the attackers to write arbitrary values to function pointers used as jump targets by the program, thereby redirecting the execution to the attacker’s executable bytes of choice. A common approach is to overwrite the Global Offset Table that holds absolute addresses to functions in dynamically linked libraries on Linux systems [116].

5.2 Related Work

5.2.1 Randomization

One of the oldest and most deployed defenses against ROP attacks is Address Space Layout Randomization (ASLR) [3]. ASLR randomizes the addresses where different program segments are loaded into memory, stripping the attacker from the ability to precisely identify where desired sequences of instructions are located in the address space of target processes. While it is a promising idea, actual deployments are far from perfect. It has been shown that ASLR can be bypassed using various techniques such as information disclosure vulnerabilities [114, 138, 149], NOP sprays [75], and brute-force attacks [153]. Additionally, it is not uncommon to find process modules that are not protected by ASLR, thereby voiding the protection of the whole process [138].

5.2.2 Gadget Elimination

Another promising approach is to produce “gadget-less” binaries. G-Free [128] was proposed as a compiler extension to eliminate usable unindented instruction sequences by realigning the generated assembly code. It protected return and jump targets by encrypting them with a random cookie at runtime. Additionally, it added checks before indirect jumps and calls to test if they were reached through a valid function entry point. Li et al. [117] proposed a similar approach to protect kernels
from ROP attacks. They proposed to eliminate all unindented \texttt{ret} instructions and to add another level of indirection over all \texttt{call} and \texttt{ret} instructions. When a \texttt{call} is executed, rather than pushing the actual return address on the stack, an index in a return address table is pushed. When a \texttt{ret} is executed, the stored index is checked against the entries in the return address table.

Both solutions are as strong as the weakest module in the process. As with ASLR, every executable module in a program has to be recompiled to enable these defenses. And manual modification of the kernel assembly code is required by [117].

5.2.3 Control Flow Integrity

Abadi et al. [40] introduced Control Flow Integrity (CFI) to prevent control flows not designated by a program. The idea is to extract a Control Flow Graph (CFG) from the program and enforce the CFG at runtime. CFI theoretically offers the best security against code-reuse attacks; however, it was not widely deployed in practice because of several reasons: 1) The overhead incurred by CFI is rather large. 2) Building a full CFG requires source code or debug information that is typically unavailable for commercial software. 3) It is practically impossible to statically construct a sound and complete CFG due to indirect control flow transfers, resulting in over-approximation of control flow targets. Recent approaches [175, 176] attempted to address these issues by enforcing coarse-grained CFI; however, it has been shown that ROP is still possible with coarse-grained notions CFI in place [69, 87].

5.2.4 Binary Rewriting and Instrumentation

Multiple solutions were presented that used binary rewriting and dynamic instrumentation to detect ROP attacks. ROPDefender [70] enforced call-ret pairing by maintaining a shadow stack of \texttt{call} and \texttt{ret} targets. ROPDefender compared the shadow stack to the actual system stack before executing \texttt{ret} instructions and raised an alarm if the two stacks did not match. ROPStop [99] used static binary rewriting to insert instrumentations that check 1) if the program counter points to a valid indented instruction, and 2) if the call stack height is valid. The 2nd constraint was checked by analyzing the CFG and computing the set of all possible call stack heights from function entry points to branching points. A ROP is detected if any of the constraints is not satisfied.
Similarly, ROPGuard [83] checked several constraints over the call stack at entry points to system calls (e.g., `ret` instructions must be preceded by `call` instructions and the `call` instruction must lead back to current entry point). These solutions are easy to deploy and require no system modifications, yet they are limited by several factors. Using CFGs is limited by the speed and accuracy of binary disassembly and CFG construction. Frame pointers (required by ROPGuard to traverse the stack) are usually omitted by compilers during optimization. Finally, call-ret pairing restricts valid call-without-return assembly constructs such as using `[setjmp; ...; longjmp;]` for exception handling and `[call; pop;]` for retrieving the program counter.

### 5.2.5 Hardware Branch Tracing

Recent ROP defenses that leverage existing hardware branch tracing features were introduced. kBouncer [129] used the Last Branch Record (LBR) on modern Intel processors to check for long sequences of consecutive call-ret instructions. (The LBR stores the most recent four to 16 indirect branches executed by the processor.) KBouncer checked at the entry of every system call if ret targets were preceded by call instructions and there was no call-ret sequence of length greater than eight. ROPecker [63] extended kBouncer by checking at arbitrary points during the program execution, counting the number of potential gadget-like sequences ahead of the program counter. Similarly, Eunomia [171] utilized the Branch Trace Store (BTS) to check for unpaired call-ret sequences. The key advantage of these approaches is their negligible overhead. Unfortunately, they have been shown to be easily bypassable [58, 69, 88].

### 5.2.6 Anomaly-based Solutions

Anomaly-based solutions learn a baseline of normal (clean) behavior and detect attacks by measuring statistical deviations from the normal behavior. These solutions have the advantage of being able to protect against a broad spectrum of attacks — including zero-day attacks — albeit tolerate some false positives and false negatives.

Krugel et al. [110] introduced an application-specific approach that used network traffic to detect malicious activities. Mazeroff et al. [123] described methods for inferring and using probabilistic models for detecting anomalous sequences of API calls. Jyostna et al. [105] proposed a system for
detecting anomalous program behavior by clustering critical system calls. Network traffic and system call defenses are simple and easy to deploy, but they are susceptible to mimicry attacks [106, 130].

One of the first works on using hardware architectural characteristics of programs was the work of Malone et al. [121]. They showed that hardware performance counters (HPC) could be utilized to detect unauthorized software changes by recording HPC measurements of the original programs and using linear regression to detect HPC deviations at runtime. Demme et al. [72] ported the idea to Android and proposed hardware modifications to detect malware using HPCs for good and malicious samples. Stewin et al. [158] proposed detecting DMA attacks by monitoring the number of transactions on the memory bus.

Tang et al. [159] combined microarchitectural characteristics with architectural characteristics to detect drive-by attacks. They assumed that attacks consists of three stages: ROP stage disables DEP, stage 1 downloads a malicious program, and stage 2 executes the malicious program. By training a one-class Support Vector Machine (oc-SVM) over the architectural and microarchitectural characteristics of benign samples, they showed that stage 1 of the attacks could be detected with high accuracy. In [46, 131], two solution were presented that trained a two-class SVM using the architectural characteristics of both clean executions and attacks.
Chapter 6
Detecting ROP with Statistical Learning of Program Characteristics

Research in ROP defenses has become an arms race. Emerging defenses are countered by new subtle variations of ROP attacks. Defenses can be defined in two broad categories: The first category attempts to prevent ROP attacks at compile time by eliminating gadgets from binaries [128] or enforcing Control-Flow Integrity (CFI) [40]. The second category aims to detect ROP attacks at runtime by monitoring the execution of programs [63,70,72,121,129,159].

Defenses in the second category can further be classified based on the detection approach into signature-based and anomaly-based. Signature-based solutions detect ROP attacks by identifying static signatures (patterns) in the execution trace of programs. The most common method is to detect gadgets execution by enforcing predefined constraints over the program counter and the call stack, either through dynamic instrumentation [70,83,99] or by leveraging existing hardware branch tracing features [63]. These solutions incur very low overhead, but the employed signatures are often incomplete due to strong constraints on the ROP structure that limit the detection capacity against new attack variants [58,69,88].

On the other hand, anomaly-based detection learns a baseline of normal (clean) behavior and detects attacks by measuring statistical deviations from the normal behavior. This approach has the significant advantage of being able to protect against unknown attack variants. Until recently, anomaly-based approaches have only leveraged software characteristics such as network traffic and system call sequences [105,123]. Meanwhile, attacks have increased in complexity, becoming stealthier and harder to detect.

Researchers have explored the potential of using hardware characteristics such as instruction mixes and branch prediction rate to detect stealthy ROP attacks [72,121,129,131,159]. Hardware...
characteristics are favored over software characteristics since it is harder for attackers to gain sufficient control over the hardware to evade detection. For example, it is easy to craft ROP payloads that mimic the behavior of clean software execution by chaining gadgets that invoke benign sequences of system calls and still execute the attack payload. In contrast, it is very challenging to craft payloads that attack the system while maintaining precise control of, for example, the branch prediction rate of the hardware. The branch prediction rate is challenging to control because attacks by definition have to go against the normal flow of the program, inevitably resulting in misprediction of branches and returns by the hardware branch predictor.

Prior work that utilized hardware characteristics used two classes of characteristics: 1) Architectural characteristics that are dependent on the instruction set architecture (ISA) such as the number of load and store instructions retired. 2) Microarchitectural characteristics, meaning characteristics that depend on the underlying microarchitecture configurations such as branches misprediction rate and cache misses. These characteristics were measured by reading the hardware performance counters (HPC) of the underlying processor. However, a common pitfall is that characteristics measured using HPC may actually hide the underlying program behavior, making the HPC-based metrics appear similar for inherently different behaviors [94, 170].

In this chapter, I introduce EigenROP [77] as a novel system for detecting ROP attacks. For the first time, I study the feasibility and value of using microarchitecture-independent program characteristics for the detection of ROP attacks. I present a novel type of anomaly-based ROP detectors that leverages microarchitecture-independent program characteristics (e.g., memory reuse distance [178], register traffic load [82], memory locality [112]) in addition to traditional hardware characteristics (see Section 6.3).

EigenROP employs a novel anomaly detection algorithm that builds on concepts from directional statistics. The fundamental idea is that strong relationships among the different program characteristics will appear as principal axes in some high-dimensional space. Since ROP executes against the control flow of the program, it is reasonable to assume that it causes some unexpected changes in the relationships between the program characteristics learned from benign runs. These changes can be detected as statistically significant deviations in the directions of the axes in the high-dimensional space. I investigate if and to what extent ROP causes changes in program characteristics, and quantify
these changes using extensive experiments with multiple in-the-wild ROP payloads and payloads generated by the ROPC ROP compiler.

EigenROP operates in two phases: a learning phase and a detection phase. During the learning (offline) phase, EigenROP collects different characteristics from a target program by monitoring its execution over benign inputs. The characteristics are measured periodically every $N$ instructions retired. A model is then constructed using Kernel Principal Component Analysis (KPCA) [147] and directional statistics (see Section 6.4). EigenROP uses a temporal model that takes into account both the current snapshot of characteristics and the history. In the detection phase, EigenROP monitors the execution of the target program and tests for deviation from the trained model.

I implemented a prototype of EigenROP on Linux using the dynamic instrumentation framework Pin [120]. I conducted several experiments to quantify the accuracy of EigenROP, the effect of involved parameters, and the incurred performance overhead (see Section 6.6). In my experiments, microarchitecture-independent characteristics resulted in 11% increase on average in detection accuracy relative to using only microarchitectural characteristics. EigenROP achieved an overall accuracy of 81%, 80% true positive rate, and only 0.8% false positive rate. The incurred performance overhead decayed exponentially as the sampling interval increases, faster than the deterioration in accuracy.

To summarize, this chapter makes the following contributions:

- Studies the effectiveness of combining microarchitecture-independent program characteristics with typical hardware characteristics for the detection of ROP attacks.
- Presents a novel anomaly detection algorithm using directional statistics of program characteristics embedded in high-dimensional space.
- Presents and discusses EigenROP as a working prototype of the presented approach.
- Demonstrates the security effectiveness of EigenROP using in-the-wild ROP attacks against common Linux programs.
- Quantifies the runtime accuracy-performance tradeoff of EigenROP.
Figure 6.1: Workflow of EigenROP. EigenROP periodically interrupts the monitored process to measure the characteristics. It embeds each window of measurements into a high-dimensional space and extracts the principal directions in that space. Then, in the learning phase, it computes a representative (mean) direction and estimates the density of distances of all principal directions to the mean direction. In the detection phase, the principal directions of incoming measurements are compared to the mean direction for significant deviation.

6.1 Microarchitecture-independent Characteristics

Microarchitecture-independent characteristics are program characteristics independent of a given microarchitecture but unique to a given instruction set architecture (ISA) and a given compiler. The characteristics are invariant of the underlying hardware internals such as cache and pipeline sizes, branch predictors size and algorithm, number of cores and their configurations.

It has been shown that microarchitecture-independent characteristics have higher discrimination power between different inherent program behaviors compared to architectural and microarchitectural characteristics [94, 170]. While characteristics dependent on the ISA, i.e., architectural characteristics, can be regarded as a subset of microarchitecture-independent characteristics, I keep them distinct in this work as is the trend in prior program characterization work [94, 121, 159, 170].

The main downside of using microarchitecture-independent characteristics is that it requires runtime instrumentation to measure the characteristics. However, the overhead decays over time as more efficient algorithms and tools are developed [51].
6.2 Overview of EigenROP

The key idea of EigenROP is to identify anomalies in program characteristics due to the execution of ROP gadgets. In this context, it is difficult to precisely define what anomalies are since that depends on the characteristics of both the monitored program and the ROP. However, it is reasonable to assume that some unexpected change occurs in the relationships among the different program characteristics due to the execution of the ROP. By extracting and learning arbitrary relationships among the program characteristics, EigenROP detects ROP by looking for unexpected changes in the learned relationships.

Given this definition of anomaly, strong relationships among the measured program characteristics should appear as principal directions in some high-dimensional space [147]. Such directions can be extracted using Kernel Principal Component Analysis (KPCA) [147]. Specifically, the principal component vectors of the measurements mapped into the high-dimensional space can be interpreted as the relationships among the program characteristics.

The general workflow of EigenROP is illustrated in Figure 6.1. First, the target program is loaded and executed. During execution, EigenROP takes a snapshot of the different program characteristics every $N$ instructions retired. Each snapshot is a $d$-dimensional vector of characteristics. The snapshots are pushed to a buffer that EigenROP iterates over using a sliding window.

In the learning phase, the target program is executed over benign inputs. For each window of measured characteristics, EigenROP maps the measurements into a high-dimensional space and extracts the principal components of the measurements in that space. It then estimates a representative direction from all the principal components and estimates the density of the distances of all principal components around that direction. Recall, the idea here is that any strong relationships among the measured characteristics will appear as principal components in the high-dimensional space. In the detection phase, EigenROP computes the distances of the principal components of incoming measurements in the high-dimensional space to the representative direction. An alarm is raised if the distance exceeds some threshold.

In the following, I define the characteristics used by EigenROP and explain in detail how learning and detection work.
6.3 Which Characteristics to Measure?

To choose the most relevant characteristics for ROP detection, I conducted several experiments to collect clean and infected measurements from a variety of programs and exploits (see Section 6.6.3). I considered most of the characteristics used in previous program characterization work [94, 121, 159, 170]. Then, I used the Fisher Score to quantify the discriminative power of each characteristic. The following is the shortlisted categories of measured characteristics. The letters between brackets denote the type of the characteristics: Architectural [A], Microarchitecture-Independent [I], and Microarchitectural [M]. Note that all the characteristics used in this work are computed in software.

- **Branch predictability [M]**. Since ROP attacks disturb the normal control flow of execution, they may increase the number of mispredicted branches by the processor branch predictor.

- **Instruction mix [A]**. This is a traditional architectural characteristic that measures the frequency of different classes of instructions (branch, call, stack, load and store, arithmetic, among others). Since ROP attacks depend on chaining blocks of instructions that load data from the hijacked program stack to registers and for returning to the stack, they may exhibit different usage of ret and call instructions as well as stack pop and push instructions.

- **Memory locality [I]**. Given a set of instructions, memory locality is the difference in the data addresses between subsequent memory accesses [112]. It is typical that a distinction is made between memory reads (loads) and writes (stores). Since ROP attacks depend on chaining gadgets from arbitrary memory locations, the attacks may exhibit low memory locality when compared to clean execution. The memory distance between subsequent reads and writes may indicate the execution of a ROP attack.

- **Register traffic [I]**. Two useful register traffic characteristics can be measured [82]: 1) The average number of register input operands to an instruction. 2) The register reuse distance, i.e., the number of instructions between writing a register and reading it. ROP attacks load data from the hijacked stack to registers typically using pop instructions that take a single operand. The number of instruction operands could be an indicator of the presence of a gadget chain. The usage degree of the registers themselves could be different from that of clean execution.
• **Memory reuse** [1]. This is an important metric that characterizes the cache behavior of programs. It measures the number of unique cache blocks referenced between subsequent memory reads [178]. For each memory read, the corresponding cache block is retrieved (assuming LRU cache). For each cache block, the number of unique cache blocks accessed since the last time it was referenced is determined. Since ROP attacks operate by using the stack for chaining gadgets spread out across the memory of the program, they shall exhibit abnormal reuse of the same memory blocks when compared to clean execution.

Table 6.1 shows the top 15 characteristics ranked by their Fisher scores. For each characteristic $i$, its Fisher Score is computed by:

$$score_i = \frac{m^+(x_i) - \bar{x}_i}{\frac{m^+(s_{i^2})}{m^+(y)} + m^-(s_{i^2})},$$

(6.1)

where $(+)$ and $(-)$ are the infected and clean classes of measurements, respectively; $\bar{x}_i^{(y)}$ and $s_{i^2}^{(y)}$ are the mean and variance of characteristic $i$ in class $y \in \{+, -\}$, and $\bar{x}_i$ is the overall mean of feature $i$ over both the infected and clean measurements. The Fisher Score is a widely established feature filtering method that assigns higher scores to features that result in greater separation between the means of clean and infected samples. I used infected and clean measurements to quantify the discriminative power of the selected characteristics. The infected measurements are not used during the learning phase of EigenROP.

Some of the scored characteristics might be redundant since the Fisher Score ignores mutual information. Therefore, I picked ten features out of the top 15 as follows: First, I excluded Instruction Level Parallelism (a measure of how many instructions of a program can be executed in parallel) since it added significant performance overhead and is highly dependent on the type of application. For example, cryptography applications may exhibit low instruction level parallelism, while a scientific computation program may exhibit high parallelism. Similarly, I excluded INST_LOAD and INST_-ARITH. Via experimentation, I decided to exclude REG_REUSE as it did not increase the accuracy of the model.
Table 6.1: Top 15 characteristics sorted by discrimination power (highest to lowest). Chosen characteristics are marked with ∗. Types A, I and M stand for “architectural,” “microarchitecture-independent” and “microarchitectural,” respectively. All counts are for instructions (insns) retired.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>INST_RET</td>
<td># leave and ret insns.</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>INST_CALL</td>
<td># near call insns.</td>
</tr>
<tr>
<td>3</td>
<td>I</td>
<td>MEM_REUSE</td>
<td>Memory reuse distance.</td>
</tr>
<tr>
<td>4</td>
<td>A</td>
<td>INST_STACK</td>
<td># pop and push insns.</td>
</tr>
<tr>
<td>5</td>
<td>I</td>
<td>MEM_RDIST</td>
<td>Memory read distance.</td>
</tr>
<tr>
<td>6</td>
<td>A</td>
<td>INST_LOAD</td>
<td># memory read insns.</td>
</tr>
<tr>
<td>7</td>
<td>I</td>
<td>REG_OPS</td>
<td>Avg. # register operands.</td>
</tr>
<tr>
<td>8</td>
<td>M</td>
<td>MISP_CBR</td>
<td>Mispredicted branches.</td>
</tr>
<tr>
<td>9</td>
<td>A</td>
<td>INST_ARITH</td>
<td># arithmetic insns.</td>
</tr>
<tr>
<td>10</td>
<td>M</td>
<td>MISP_RET</td>
<td>Mispredicted ret insns.</td>
</tr>
<tr>
<td>11</td>
<td>A</td>
<td>INST_STORE</td>
<td># memory write insns.</td>
</tr>
<tr>
<td>12</td>
<td>I</td>
<td>MEM_WDIST</td>
<td>Memory write distance.</td>
</tr>
<tr>
<td>13</td>
<td>A</td>
<td>INST_NOP</td>
<td># nop insns.</td>
</tr>
<tr>
<td>14</td>
<td>I</td>
<td>REG_REUSE</td>
<td>Register reuse distance.</td>
</tr>
<tr>
<td>15</td>
<td>I</td>
<td>ILP</td>
<td>Instruction level parallelism.</td>
</tr>
</tbody>
</table>

6.4 Learning and Detection

Given a sequence $T$ of $d$-dimensional measurements, divide $T$ into $n$ subsequences using a sliding window of width $m$. Let us denote the resulting subsequences by:

$$S^{(j)} = \begin{bmatrix} x_1^{T(j)} \\ x_2^{T(j)} \\ \vdots \\ x_m^{T(j)} \end{bmatrix},$$  \hspace{1cm} (6.2)

for $j = 1 \ldots n$. Each $x_i^{(j)}$ is a vector of $d$ measured characteristics.

Next, each $S^{(j)}$ is embedded (implicitly mapped) into a higher dimension space $H$ with $\Phi : \mathbb{R}^d \rightarrow H$ and the principal component vectors of $S^{(j)}$ in $H$ are extracted. This is done using Kernel PCA [147], solving the following eigenvalue problem:

$$\lambda_i^{(j)} v_i^{(j)} = K v_i^{(j)},$$  \hspace{1cm} (6.3)

where $\lambda_i^{(j)}$ are the eigenvalues of $K$, $v_i^{(j)}$ are the normalized eigenvectors of $K$, and $K$ is the $m \times m$ kernel matrix $[k(x_i^{(j)}, x_l^{(j)})]$ for $i = 1 \ldots m; l = 1 \ldots m$. Here, $k$ is the Radial Basis Function.
(RBF) kernel given by:

\[ k(x_1, x_2) = \Phi(x_1)\Phi(x_2)^T \]  \hspace{1cm} (6.4)

\[ = \exp\left(-\gamma \|x_1 - x_2\|^2\right), \]  \hspace{1cm} (6.5)

where \( \gamma = \frac{1}{d} \) and \( K \) is centered \([147]\), i.e., \( K = K - 1_mK - K1_m + 1_mK1_m \) where \( 1_m \) is an \( m \times m \) matrix for which each element takes the value \( \frac{1}{m} \).

Using the eigenvalues and eigenvectors in \( \mathcal{H} \), the resultant direction \( v^{(j)} \) of the data \( S^{(j)} \) embedded in \( \mathcal{H} \) is then computed by:

\[ v^{(j)} = c \sum_{i=1}^{m} \lambda_i^{(j)} v_i^{(j)}, \]  \hspace{1cm} (6.6)

where \( c \) is a normalizing factor such that \( v^{(j)^T}v^{(j)} = 1 \). This direction can be perceived as a representative direction of all the principal axes of \( S^{(j)} \) in the kernel space \( \mathcal{H} \).

The mean direction \( \mu \) of \( T \) can be computed by:

\[ \mu = \frac{\sum_{j=1}^{n} v^{(j)}}{\left|\sum_{j=1}^{n} v^{(j)}\right|}, \]  \hspace{1cm} (6.7)

The direction \( \mu \) is the representative direction for the entire trace of characteristics. Since the extracted directions \( v^{(j)} \) are distribute around \( \mu \), we can construct the following similarity vector \( Z \):

\[ Z = \begin{bmatrix} v^{(1)^T} \mu \\ v^{(2)^T} \mu \\ \vdots \\ v^{(n)^T} \mu \end{bmatrix}, \]  \hspace{1cm} (6.8)

where each row corresponds to the angular distance between each direction \( v^{(j)} \) and \( \mu \).

Next, a kernel density is estimated over \( Z \) using the standard normal kernel density estimator:

\[ f_h(z) = \frac{1}{nh} \sum_{i=1}^{n} N\left( \frac{z - z_i}{h} \right), \]  \hspace{1cm} (6.9)
where \( h \) is the smoothing parameter (the bandwidth), \( z_i \in Z \), and \( N \) is the standard normal function. I chose the value of \( h \) using grid search in my implementation.

It is expected that the resulting density will be close to exponential since the directions extracted from clean measurements are expected to be concentrated (tightly distributed around \( \mu \)), resulting in a skewed density with a peak around high similarity values. We can reduce the skewness of \( f_h \) by applying the following logarithmic transform:

\[
\hat{f}_h(z) = f_h(z) \log(f_h(z)), 
\]

where the area under the curve of \( \hat{f}_h(z) \) gives the entropy \( \eta \) of \( \hat{f}_h \). This transforms the bulk of the density towards the peak, resulting in a shorter tail (easier to threshold).

This concludes the learning phase. The following subsection explains the anomaly metric and the detection phase of EigenROP.

### 6.4.1 Anomaly Metric

Given an incoming subsequence of measurements \( S^{r(j)} \), an anomaly is detected if the direction of \( S^{r(j)} \) in the \( H \) space is significantly different from the learned directions around \( \mu \). The decision \( r \) is computed by:

\[
v^{(j)} \text{ from Equation (6.6)} \] (6.11)

\[
z^{r(j)} = v^{(j)T} \mu \] (6.12)

\[
\zeta = \int_{-1}^{z^{r(j)}} \hat{f}_h(z) \, dz \] (6.13)

\[
r = \text{sgn}(\zeta - \theta \eta) , \] (6.14)

where \( \theta \in (0, 1) \) is the detection threshold that sets the fraction of the entropy the model leaves out for detecting attacks. This concludes the detection phase.
The following summarizes the steps taken by EigenROP in the learning and detection phases:

**Learning Phase**

1. Periodically, collect program characteristics \( \{ S^{(j)} \}_{j=1}^n \) of the target program.
2. Extract the principal directions \( \{ v^{(j)} \}_{j=1}^n \) in a higher-dimension kernel space.
3. Compute a representative direction \( \mu \) from \( \{ v^{(j)} \}_{j=1}^n \).
4. Estimate \( \eta \) of the distance between the principal directions and \( \mu \).

**Detection Phase**

5. Repeat steps 1 and 2.
6. Compute the anomaly metric \( r \). If \( r \) equals \(-1\) then an attack is present.

### 6.4.2 Detection Time and Space Complexity

Computing the anomaly metric requires performing the KPCA computation (Equation (6.3)) in \( O(m^3) \) [147]. Computing the resultant vector (Equation (6.6)) takes \( O(m^2) \). The distance in Equation (6.12) is computed in \( O(m) \). Thus, it takes a total time of \( O(m^3) \) to compute the anomaly metric. The model requires space \( m \cdot d \) for the incoming measurements window \( S^{(j)} \), \( m \) for the representative direction \( \mu \), and \( c \) for the transformed density (Equation (6.10)) where \( c \) is the number of points of the density. Thus, it takes a total space of \( O(md + c) \). All terms in the prototype implementation of EigenROP are bounded: \( d = 10, m \leq 10 \) and \( c \leq 1000 \).

### 6.4.3 Handling Multiple Runs

The algorithm discussed so far focused on a single run of the monitored program. To handle multiple runs, EigenROP proceeds as follows: Given a set \( \{ T^{(i)} \}_{i=1}^k \) of sequences where each \( T^{(i)} \) corresponds to a different run of the monitored program, compute the family of sets of directions \( \{ \{ v^{(j)} \}_{j=1}^n \}_{i=1}^k \) and compute \( \mu \) over the entire family. Storing the entire set of directions is not necessary since \( \mu \) and the distance density can be computed iteratively using streamed mean and density algorithms.
6.5 Implementation

I implemented a proof-of-concept prototype of EigenROP on top of MICA [95] (a Pintool for collecting program characteristics). The EigenROP module is implemented in ~700 lines of Python with the aid of the SciKit-Learn [27] machine learning toolkit. Pin [120] is a generic dynamic instrumentation framework with a rich API that a Pintools uses to specify its own instrumentation code. I chose Pin since it achieves the best performance among various dynamic instrumentation platforms [120].

Figure 6.2 shows the architecture of EigenROP within Pin. MICA uses the instrumentation API of Pin to specify its own instrumentation code that computes the different characteristics. As the program executes, the JIT compiler in Pin intercepts the program traces and compiles the instrumentation code into the program. The characteristics are computed over the program traces. A program trace is a chain of multiple basic blocks that end with an unconditional jump. The measurements reported by MICA are stored in a \(d\)-dimensional circular buffer one row at a time. The EigenROP module consumes and processes the buffer using a sliding window as explained in Section 6.4. Finally, the learned directions and densities are stored on disk for usage in the detection phase where the same procedure is followed in addition to computing the anomaly metric. If a ROP is detected, EigenROP logs an alarm and terminates the target process.
Table 6.2: Dataset used in the experiments.

<table>
<thead>
<tr>
<th>Program</th>
<th>Avg. Payload Length</th>
<th># of Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmp</td>
<td>800</td>
<td>80</td>
</tr>
<tr>
<td>cpio</td>
<td>650</td>
<td>210</td>
</tr>
<tr>
<td>diff</td>
<td>910</td>
<td>140</td>
</tr>
<tr>
<td>file</td>
<td>700</td>
<td>315</td>
</tr>
<tr>
<td>grep</td>
<td>631</td>
<td>150</td>
</tr>
<tr>
<td>hteditor</td>
<td>60</td>
<td>100</td>
</tr>
<tr>
<td>openssl</td>
<td>1021</td>
<td>195</td>
</tr>
<tr>
<td>php</td>
<td>400</td>
<td>265</td>
</tr>
<tr>
<td>sed</td>
<td>570</td>
<td>350</td>
</tr>
<tr>
<td>sort</td>
<td>712</td>
<td>110</td>
</tr>
<tr>
<td>stat</td>
<td>673</td>
<td>110</td>
</tr>
<tr>
<td>wget</td>
<td>813</td>
<td>90</td>
</tr>
</tbody>
</table>

Total Samples: 2115

6.6 Evaluation

I evaluate the security effectiveness, the added value of using microarchitecture-independent characteristics, and the tradeoff between runtime overhead and the detection accuracy of EigenROP. I conducted several experiments using in-the-wild ROP attacks and attacks generated by the ROPC [25] compiler. I used the UnixBench [35] systems benchmark for performance evaluation. All experiments were conducted on an Intel Core i7-4870HQ 2.5 GHZ machine with 4 GB of RAM, running 32-bit Linux Ubuntu 12.04, Intel Pin 2.14, MICA 0.40, and GCC 4.6.3.

6.6.1 Evaluation Metrics

I used Receiver Operating Characteristics (ROC) curves and the area under the curve (AUC) scores to evaluate the detection accuracy of EigenROP. The $x$-axis of the ROC curve gives the false positive rate (FPR) and the $y$-axis gives the true positive rate (TPR). The FPR (eqv. with 1 – specificity) represents the probability of false alarm, i.e., the likelihood of mislabeling a clean execution as an attack, given by $FP/(FP + TN)$. The TPR (eqv. with sensitivity) represents the probability of correct detection of ROP execution given by $TP/(TP + FN)$. Each point on the ROC curve corresponds to the FPR and TPR for a specific value of $\theta \in (0, 1)$. The area under the curve (AUC) of the ROC is also computed as a quantitative single value measure of the accuracy of the system for a variable $\theta$. The higher the AUC, the higher the detection accuracy. The AUC reaches its best value at one and its worst at zero.
6.6.2 Dataset and Evaluation Procedure

I used two publicly available ROP exploits: OSVDB-ID:87289 [12] and OSVDB-ID:72644 [22] for the Linux Hex Editor (hteditor) version 2.0.20 and PHP version 5.3.6, respectively. I also used various exploits generated by the ROP gadgets finder and compiler ROPC [25] for common Linux programs (four different exploits per program). Table 6.2 shows the programs used, the average payload length (the number of instructions) of each exploit, and the number of samples.

I collected clean samples for the target programs by running the functionality tests that shipped with each program. I ran hteditor on 100 random PDF files downloaded from the web since it did not ship with functionality tests. I collected infected samples by directly injecting code into the target process to load a given exploit payload into memory and execute it. This approach was previously used in [63,131]. The payload (gadgets) is executed by directly jumping to the beginning of the payload at random points during the execution of the process. Each payload execution was considered an infected (attack) sample.

Each program was evaluated using 5-fold cross-validation with four clean folds for training and one clean fold for testing along with infected samples. I used the same number of clean and infected samples in the testing fold. The mean of the resulting five TPRs and FPRs is then used in computing the ROC and its AUC. Labeled measurements were collected strictly for testing; EigenROP uses only the clean measurements for training.

6.6.3 Detection Accuracy

Hteditor OSVDB-ID:87289 and PHP OSVDB-ID:72644

EigenROP successfully detected the hteditor ROP exploit with sampling intervals up to 16k instructions retired and detected the PHP ROP with sampling intervals up to 32k. In both cases, EigenROP resulted in zero false positives. I emphasize that the focus here is on the detection of the ROP stage of the exploits, i.e., the execution of a gadget chain, rather than the execution of a foreign process. (Detection of a foreign process was shown to be easily detectable [121,159].) EigenROP still detected the deviation in the programs characteristics despite the very small ROP length (∼60 instructions in the case of hteditor) compared to the sampling window size.
Overall Detection Accuracy

Figure 6.3 shows the overall ROC of all experiments for a sampling interval of 16k instructions. EigenROP achieved an overall accuracy (AUC) of 81%. The best point of performance had 80% TPR and 0.8% FPR. Again, EigenROP solely focuses on the detection of ROP; relevant prior work [121,159] assumed the attacks undergo multiple stages such that only the first stage is a ROP chain while the rest are injected code or a different process. Detecting the ROP stage is significantly more challenging since the ROP chain length is usually limited (a few hundred instructions). While the authors in [121,159] detected the non-ROP stages of the attack with high accuracy, they noted that their proposed models performed poorly in the detection of the ROP chains alone (AUC ranged from 49% to 68%). In contrast, EigenROP focuses on the detection of the execution of the ROP gadget chain itself and achieved significantly higher accuracy.

Sampling Granularity

The breakdown of the detection accuracy for different sampling intervals is shown in Figure 6.4. As expected, the accuracy drops for very large sampling intervals because of the small number of instructions of the attacks. Wget had the worst detection accuracy due to its excessive use of signals, exhibiting poor locality and reuse (see Section 6.7 for discussion). The density estimate of Wget was very heavy-tailed, resulting in low discrimination between clean runs and attacks. OpenSSL had the
highest detection accuracy as its characteristics had higher concentration around the mean direction. The bulk of the distribution of the AUC curves neared the best accuracy curve (the AUC was skewed towards the worst accuracy curve), indicating that the behavior of Wget was possibly an outlier.

**Microarchitecture-independent vs. Other Characteristics**

Figure 6.5 shows the difference in accuracy with and without the microarchitecture-independent characteristics. By including microarchitecture-independent characteristics, an increase of 9% to 15% in accuracy was achieved. This indicates that microarchitecture-independent characteristics contribute significantly to the detection performance of EigenROP.

**Sliding Window Size**

Figure 6.6 shows the effect of changing the sliding window size \( m \) on the detection accuracy. The window size \( m \) controls the amount of temporal information available to the model. We can observe that the effect of the window size on accuracy goes through three stages: First, too small window sizes hurt the detection accuracy since small windows give higher variances in principal directions that lead to a higher FPR. Second, as the window size increases, the detection accuracy improves since the directions become more stable around \( \mu \). Finally, the accuracy deteriorates for too large window sizes since the influence of clean measurements on the principal directions dominates that of the ROP payload, resulting in a lower TPR.
Figure 6.5: AUC with and without the microarchitecture-independent characteristics.

Figure 6.6: AUC for different sliding window sizes. Both too small and too large windows result in lower detection accuracy.
6.6.4 Overhead-Accuracy Tradeoff

I quantified the overhead of EigenROP for different sampling intervals by measuring the overall percentage slowdown in execution of UnixBench [35]. Figure 6.7 shows the overhead and accuracy tradeoff. The overhead incurred by EigenROP exponentially decreased as the sampling interval increased. The reduction in overhead outpaced the decay in accuracy. The overhead incurred by MICA was approximately constant as the total number of instructions analyzed by MICA is invariant of the sampling interval. Overall, the incurred runtime overhead is comparable to similar dynamic instrumentation and HPC-based defenses [70, 131, 159]. I did not perform any optimization attempts to reduce the overhead of EigenROP or MICA. This work is orthogonal to how the program characteristics are collected. While I used MICA and Pin in the prototype implementation of EigenROP, they may not be the best tools for full build-out and full production. Finally, the memory and space overhead incurred by EigenROP are bounded and negligible (see Sections 6.4.2 and 6.5).

6.7 Discussion and Improvements

6.7.1 False Positives and Negatives

The detection approach of EigenROP (and relevant HPC-based solutions [72, 121, 159]) is based on the hypothesis that programs exhibit characteristics that are relatively concentrated around some
statistic — in our case, the mean direction. However, if a program exhibits behavior that has a large spread, it becomes harder to separate anomalies from benign executions, resulting in a higher false positive rate (or a lower true positive rate).

From my experience with EigenROP, I observed that programs that use far jumps (e.g., `setjmp`, `longjmp`, `signal`) or extensively multiplex between data sources (e.g., using `select` for socket multiplexing) are more likely to suffer from false positives. The reason is that these programming constructs access far code and data, exhibiting poor branch predictability, memory locality, and reuse. A possible workaround is to identify the entry and exit points of such code sites and build a separate model for the characteristics exhibited by those code sites. ROP chains missed by EigenROP were very short chains (<40 instructions) with small gadgets (two to four instructions per gadget). They were not detected due to the large sampling interval compared to their chain lengths. To handle these very short chains, EigenROP can be complemented by low-overhead solutions that target short gadgets and chains (e.g., kBouncer [129] and ROPecker [63]).

### 6.7.2 ROP Variants

In the evaluation of EigenROP, I used conventional ROP payloads that use return instructions to chain the gadgets. However, several variants of ROP were discovered by researchers. For example, in [50], Jump-Oriented Programming (JOP) was introduced where indirect jumps are employed to chain the gadgets rather than using return instructions. JOP simulate `ret` using a gadget that pops an address from the stack then jumps to that address using an indirect jump instruction, i.e., a pop-jump gadget. To use the pop-jump gadget, other gadgets have to end in an indirect jump that transfers control to the pop-jump gadget, e.g., `[add; mov; ...; jmp eax; pop ebx; jmp ebx;]` where `[jmp eax;]` jumps to the pop-jump gadget, and `[pop ebx; jmp ebx;]` executes the pop-jump gadget and transfers control to the next gadget.

In EigenROP, I picked the characteristics that cover the behavior of all ROP variants (branches, calls and returns, memory locality and reuse, stack usage, and `nop` sleds) regardless of how the gadgets are chained. Also, it is easy and straightforward to include other relevant characteristics if needed, such as the number of indirect jump instructions retired. Overall, EigenROP is robust against
attack variations since it captures the execution behavior of benign runs and does not put strong assumptions on how the gadgets are chained at the ISA level.

6.7.3 Evasion and Mimicry Attacks

Three recent attack gadgets were presented [58] that bypass ROP defenses through evasion and mimicry: call-preceded gadgets, evasion gadgets, and history-flushing gadgets.

Call-preceded gadgets are constructed from sequences of instructions that are preceded by a call instruction in the program memory. Such gadgets violate the assumption made by the majority of defenses [63, 70, 99, 129]: A sequence ending in ret must be legitimate if it was preceded by any call. Since EigenROP does not depend on branch tracing, it is not vulnerable to attacks based on call-preceded gadgets. Moreover, the return address will be mispredicted (regardless of the gadget type) unless the call-ret are strictly paired. Since EigenROP takes the misprediction rate of returns into account (see Section 6.3), call-preceded gadgets will result in abnormal mispredictions, potentially increasing the detection accuracy.

Evasion gadgets were introduced for evading ROP detectors that use heuristics based on the length of the gadget chain (e.g., [63, 129]). These detectors detect ROP by identifying gadget chains within some window of the execution trace. The heuristics are based on the length of the gadgets within the chain, presuming that short gadgets are likely part of an executing ROP. Evasion gadgets violate that presumption by using long enough gadgets. Since EigenROP does not depend on the gadget chain length, rather on the characteristics of the gadgets, it is not vulnerable to attacks based on evasion gadgets.

History-flushing gadgets target defenses that only keep a limited history about execution. (History length depends on the size of the hardware buffer that stores the history.) History is flushed by utilizing innocuous gadgets to fill up the history. For example, kBouncer [129] uses the most recent 16 taken branches recorded in the processor’s Last Branch Record (LBR). While kBouncer is very efficient against short ROP chains, it can be evaded by a ROP chain that executes any 16 valid indirect jumps to completely fill the LBR with legitimate branches [58].

In our context, flushing the history means manipulating all affected characteristics by the ROP such that they appear normal. The attacker would need to chain gadgets that achieve the attack goal
yet exhibit similar characteristics to benign code. This is arguably hard to accomplish in practice. Chaining more gadgets requires larger attacker-controlled memory space. Should the attacker include benign code in the ROP to mimic normal behavior, the benign code would be required to either have no impact on the actual ROP execution or have its impact undone by chaining more gadgets. As noted in [58, 131], history flushing comes at the expense of significant slowdown (reported 20-times slowdown) in the execution of the ROP payload.

Randomization has been proposed as a defense against evasion and mimicry attacks in anomaly-based intrusion detection systems [166,169]. Recently, Smutz and Stavrou [155] showed that mimicry attacks could be efficiently detected by judging the quality of detection using an ensemble of classifiers. A potential defense strategy for EigenROP is to randomize the set of measured characteristics and build multiple detectors using different subsets of characteristics. The detectors can be constructed using different models where a subset of the models is chosen at runtime at random. The models can also be randomized at different points in the program. For example, using five out of 15 characteristics to build five models gives \(5 \cdot \binom{15}{5} = 15015\) possible configurations. Since the attacker does not have direct control over the program characteristics, she would need to craft ROP payloads that bypass all possible configurations of detectors and characteristics, significantly increasing the cost to attack.

6.7.4 Overhead Reduction

The current downside of using microarchitecture-independent characteristics is the need for dynamic instrumentation to compute the characteristics. As shown in Section 6.6.4, this may incur a non-negligible overhead penalty. Efficiently computing software characteristics is an active research area and more efficient program characterization algorithms and tools are being developed [51]. The need for dynamic instrumentation can be eliminated if the hardware or the kernel provide support by computing the required characteristics. Rather than instrumenting the process in user space, the characteristics can be computed (by the kernel or the hardware) and written to a memory-mapped ring buffer that is readable in user space. In case the buffer is not consumed quickly enough, an interrupt can be triggered to pause the monitored process. A similar approach is adopted by the Linux performance counter subsystem [15] which already provides support for a wide range of architectural and microarchitectural characteristics.
6.7.5 Input Coverage

In the learning phase of EigenROP, the target program is executed over benign inputs. Sufficient input coverage could arguably be a challenging task for the deployment of EigenROP. In my evaluation, I used the positive functionality tests that shipped with the programs to train EigenROP. Functionality tests are integral to the software development lifecycle. A possible alternative is to train EigenROP using successful dry runs during internal acceptance and pre-release testing. Additionally, EigenROP can even be trained by end users. To avoid learning bad behavior, the learned models can be aggregated from clusters of users and averaged (by computing the mean directions and densities) then filtered (cleaned) from outliers. Further, EigenROP can continue learning even after deployment by iteratively updating the learned directions and densities. This can be a privilege that is tied to the user group (e.g., update the models only from processes owned by admin users).

6.8 Summary

I presented EigenROP, a novel anomaly-based ROP detector that utilizes program characteristics and directional statistics. To the best of my knowledge, this is the first study of the effectiveness of using microarchitecture-independent program characteristics versus typical architectural and microarchitectural characteristics in the detection of ROP. I demonstrated the ability of EigenROP to detect both in-the-wild and pure ROP exploits despite the short ROP payload length. EigenROP is unsupervised, fully transparent, and does not require any side information about the protected programs. One limitation of using microarchitecture-independent characteristics is the need for dynamic instrumentation to collect the measurements. One potential avenue to significantly reduce the overhead is by implementing the run-time monitors in hardware. Hardware support would also help increase the detection accuracy by enabling low-cost fine granularity monitoring. While this work demonstrates that ROP payloads can be detected using simple program characteristics, there are still needed improvements concerning detection accuracy and overhead reduction. Despite that, EigenROP raises the bar for ROP attacks and can be easily coupled with hardware-based defenses to detect ROP transparently without program changes [96, 113].
Chapter 7
FRA: Background and Related Work

While the majority of code-reuse defenses focus on attacks that reuse unintended instruction sequences (e.g., ROP), attack variants that reuse whole functions, i.e., Function-Reuse Attacks (FRA), have received little attention primarily because defending against FRA is a challenging task requiring to differentiate between legitimate and malicious calls of the same function.

Presently, memory subversion remains an unsolved security threat. By manipulating control data, such as function pointers and return addresses, attackers can hijack the control flow of programs and execute arbitrary code. Even though modern systems are equipped with W\(\times\)X and Data Execution Prevention (DEP), attackers can still achieve arbitrary code execution by repurposing existing code from the program memory, in what is known as code-reuse attacks. This can range from reusing blocks of instructions, such as Return Oriented Programming (ROP), to even reusing whole functions in a Function Reuse Attack (FRA).

The use of Control Flow Integrity (CFI) [40], which is a critical program security property, can assure the program does not execute unintended code. Unfortunately, constructing a sound and complete CFI policy has proven to be a challenging task [58]. Enforcing CFI is especially hard due to indirect control flow transfer, such as indirect calls through function pointers. The problem becomes even harder if the source code is not available. This makes binary-only solutions very desirable, since, in practice, the source code of many programs is not always available, and that includes many commercial products, 3rd party libraries, legacy software and firmware to name a few. Even if the source code is available, compiling in new protections is not always feasible or desirable, for instance, due to the presence of legacy code and compiler dependencies.
7.1 FRA in C++ Binaries

Indirect calls are prevalent in OOP languages in order to enable polymorphism. Of particular interest to us is C++, where all major compilers, including GCC, LLVM, and MSVC, support C++ polymorphism via tables of function pointers. This is also the case for compilers of closely related languages, such as C# and D. C++ supports class and function polymorphs by allowing derived classes to redefine base functions that are declared virtual. Each object of a class that (re)defines virtual functions stores a pointer (vptr) to a read-only table of pointers to virtual function definitions (called vtable for short). To invoke a virtual function, the compiler generates code that indirectly executes the corresponding function in the object’s vtable (see Section 8.1). I refer to such code sites in the binary as virtual call (vcall) sites.

A virtual table (vtable) is a reserved read-only memory section in the binary that contains function pointers to the bodies of virtual methods implemented by a class. Each virtual method in a class has a corresponding offset in the class vtable, the address at which points to the actual implementation body of the method in the code section. Whenever an object invokes a virtual method, the object’s class vtable is accessed, and the address at the corresponding method offset is loaded and invoked (indirect call). If an object has virtual methods, a pointer (called vtable pointer) is stored in the object’s structure that points to the object’s class vtable. The vtable pointer is typically stored as the first entry of the object structure.

In an unprotected binary, an attacker with control over an object’s memory or vtable can call any function within the program whenever the program uses the object’s vtable to make a vcall. This is typically achieved by exploiting a memory access bug that enables overwriting the vptr in an object’s memory, in what is known as a “vtable attack”. Perhaps the most common class of enabler bugs in this category is the infamous use-after-free [41]. Here, a pointer to a freed object is used in a later program statement (a dangling pointer) to invoke one of the object’s virtual functions. This dangling pointer can allow an attacker to execute arbitrary code if she can control the contents of the object’s freed memory, e.g., using heap overflows or heap spraying [68]. Such bugs are very prevalent in commodity desktop applications, such as office suites and browsers, since they are typically written in C++. Recent studies (e.g., [55,100,115]) suggested use-after-free vulnerabilities account for at least
69% of all vulnerabilities in browsers, about 50% of Windows 7 exploits, and 21% of all vulnerabilities in all operating systems.

Without loss of generality, FRA in C++ can be divided into three categories [100, 124, 133, 173]:

1. **Vtable corruption.** This is a legacy attack that overwrites the contents of legitimate vtables. The attack is prevented by modern compilers by storing the vtables in read-only memory.

2. **Vtable injection.** Here, the attacker first injects a fake vtable into the program memory. Then, she points the vtable pointer of a hijacked object to the injected vtable. The injected vtable can therefore point to arbitrary functions or gadgets in the executable program memory.

3. **Vtable reuse.** This attack operates the same way as vtable injection attacks, except that the attacker does not inject any counterfeit vtables in memory. Instead, the attacker reuses already existing vtables in the program memory.

### 7.2 Related Work

#### 7.2.1 ASLR

Address Space Layout Randomization (ASLR) [3] is perhaps the most deployed defense against code-reuse attacks. ASLR randomizes the addresses where different program sections are loaded into memory, stripping the attacker from the ability to precisely identify where desired code is located. Actual deployments, however, are far from perfect, and it has been shown that various ASLR deployments can be bypassed (e.g., [138, 149]) using various techniques, such as information disclosure vulnerabilities [114, 138, 149], nop sprays [75], and brute-force attacks [153]. In addition, it is not uncommon to find process modules that are not protected by ASLR, thereby voiding the protection of the whole process [138].

Crane et al. [65, 66], proposed randomization based defenses resilient to memory disclosure attacks. The two approaches utilized the newly introduced execute-only memory pages via the Extended Page Tables (EPT) virtualization technology in Intel processors since the Nehalem microarchitecture. Both solutions require hardware support, kernel and compiler changes, and source
recompilation. While randomization increases the attack cost by increasing the attacker's uncertainty, it only provides probabilistic guarantees.

7.2.2 Control Flow Integrity

Abadi et al. [40] introduced Control Flow Integrity (CFI), which prevents control flows not intended by the original program. The idea is to extract a Control Flow Graph (CFG) from the program and enforce the CFG at runtime. Unfortunately, CFI is not widely adopted in practice, because of two main hurdles: 1. building a complete CFG is a very challenging task, especially without access to source code or debug symbols; and 2. the overhead incurred by ideal CFI is rather large. Recent approaches [175, 176] attempted to address those issues by enforcing coarse-grained CFI. However, it has been shown [57, 58, 65, 69, 87, 148] that code reuse is still possible with such loose notions of CFI in place. Recently, PathArmor [164] showed that context-sensitive CFI can be enforced with little overhead using recent hardware features. However, it lacked forward-edge context sensitivity which made COOP attacks still possible. TypeArmor [165] enforced a generic binary-level policy based on the number of produced and consumed function arguments. As demonstrated by the analysis in Section 8.10, such policy is imprecise compared to semantic-aware policies. Nevertheless, generic CFI solutions are complementary to this work, where I only focus on protecting the integrity of vcalls.

7.2.3 Compiler Solutions

Recent versions of the GCC compiler support a new vtable verification (VTV) [160] feature, which inserts checks before each virtual method call that asserts that the vtable pointer is valid for the invoker object type. It does this by computing the set of all possible valid vtable pointers for each class hierarchy, and checking for set membership at runtime. Shrinkwrap [91] enhanced this by enforcing object-call pairing for each vcall in the program, as well as fixing corner cases that were discovered in the implementation of VTV. Similarly, SafeDispatch [100] extended LLVM to support a similar policy to VTV. Also for LLVM, VTrust [174] proposed a hash-based technique to verify the integrity of vcalls. For the MSVC compiler, VT-Guard [124] proposed a defense that inserted a secret cookie into each vtable and checked if the cookie is valid before each vcall. While this makes it harder for an attacker to inject a valid vtable, it falls short against memory disclosure attacks that can leak

102
the cookie value. Additionally, such defense does not mitigate vtable reuse attacks, as the attacker can still point the vtable pointer to any valid vtable in the program memory.

Recently, Bounov et al. [52] proposed an LLVM extension that reorders vtables such that integrity policies can test for vtable membership in constant time. In general, compiler-based solutions have the maximum visibility into the source code, allowing them to enforce stronger policies than ours. Nevertheless, they require access to the source code and recompilation of all linked modules, which may not be feasible in practice. Other solutions, such as CETS [126] and Dangnull [115], attempted to eliminate dangling pointers altogether by tracing object pointers and nullifying them upon deletion. Unfortunately, sound and complete tracing of pointers is NP-Hard [111], especially with pointer aliasing and multithreading constructs available in all modern programming languages. Additionally, there are various ways to mount vtable attacks besides using a dangling pointer, such as buffer overflow, format string, and type confusion attacks. That said, eliminating dangling pointers is complementary to this work and resembles a strong layer of defense against various memory corruption attacks.

It is important to emphasize that, while compiler-based solutions have the maximum visibility into the source code, they require recompilation of all linked modules, which is infeasible in practice as the source code is not always available (propriety code, 3rd party libraries, legacy code, etc.).

7.2.4 Binary Solutions

Multiple binary solutions were proposed to defend against vtable attacks. TVIP [86] used static analysis to identify and extract vtables and vcall sites. At runtime, it checked at each vcall site that the referenced vtable is read-only and the vcall offset is in the vtable. Similarly, RECALL [74] identified unsafe casting in MSVC binaries by matching the layouts of objects that reach vcall sites. Both solutions worked on an intermediate binary representation obtained by lifting the x86 assembly to a static single assignment (SSA) form. However, as the authors explained, this is not error-free.

VTint [173] relocated vtables to a read-only memory section, and checked before every vcall that the referenced vtable is read-only. VTint incurred low overhead, but at the same time it suffered from poor identification accuracy. For instance, VTint identified only 115 vtables and 200 vcalls for 447.dealII, whereas VCI (see Chapter 8) identified about 7 times as many. Similarly, vfGuard [133]
used static analysis to reconstruct the set of all possible targets for each vcall site, given the vcall offset, and instrumented the binary to check for membership. Unfortunately, it was assessed that such policies are not precise enough to stand against COOP attacks [65, 148].

On a different defense front, solutions were proposed to detect memory corruption and access bugs. Valgrind [127], AddressSanitizer [145], and Undangle [55] are a few examples of dynamic memory monitoring systems that help detect memory access errors, including use-after-free. However, the overhead is prohibitive for practical deployment as a security solution (25x runtime overhead). DieHard [49] provided a probabilistic memory integrity guarantee by randomizing and expanding the heap. While it incurred much less overhead than full-blown dynamic memory monitoring, it required at least double the heap size for each program it protects, which is not feasible in practice. More recently, VTPin [146] introduced a simple and novel solution by directly managing deallocations, and preventing reuse of deleted objects by repointing their vtptr to a safe vtable. For that purpose, however, it required hooking the free and malloc calls, the presence of RTTI in the binary, as well as catching segfaults that may result from probing unmapped memory.

Complementary to this work is C++ reverse engineering efforts. In Smartdec [81], the authors proposed a system to reconstruct C++ class hierarchies from RTTI. Similarly, Objdigger [101] extracted objects and member functions of classes from compiled MSVC binaries. While decompilation is very valuable for many security problems, it is tuned for vcall integrity as decompilation poses a different set of problems than vcall integrity enforcement.
Chapter 8
Strict Virtual Call Integrity Checking for C++ Binaries

In this chapter, I present VCI [79] as a static binary CFI system that retrofits C++ binaries with defenses against vtable attacks. VCI protects the binaries by enforcing a strict CFI policy that limits the number of callable function from vcall sites (see Section 8.2). VCI works on stripped binaries, without needing debug, symbol or type information. To determine valid function targets, I developed algorithms to reconstruct several C++ semantics from binaries, namely: vtables, constructors, class layouts, class hierarchies, and vcalls (see Section 8.3). VCI exploits patterns in the assembly, and uses backward slicing and inter-procedural analysis to symbolically trace the this pointer expressions of objects across function boundaries. It builds a mapping between vcall sites and their target class types. It then instruments the binary by generating and injecting the integrity policy to enforce the mapping at runtime.

I implemented a prototype of VCI in C++ on Linux, using Dyninst [11] for binary parsing and rewriting. The prototype consists of \( \sim 3500 \) SLOC for the analysis in addition to a \( \sim 500 \) SLOC dynamic library where the integrity policy procedures reside. Experimental results (see Section 8.4) on the C++ SPEC CPU2006 benchmarks and Mozilla Firefox show that VCI significantly reduces the attack surface compared to the state-of-the-art binary vtable defenses. For instance, in comparison with VTint [173] and vfGuard [133], VCI achieved at least 96\% and 48\% additional reduction in the number of allowable vcall targets, respectively. In comparison to GCC VTV [160] (source-based ground truth), VCI achieved the highest precision amongst other binary solutions, with 100\% precision in some cases and greater than 60\% precision for the majority of the test programs. My experiments show that VCI incurs a low runtime overhead (\( \sim 7.79\% \)), and can defend against real-world exploits including the recent COOP attacks [66, 148].

In summary, this chapter makes the following contributions:
• Presents VCI as a binary analysis and rewriting tool that automatically analyzes and retrofits stripped C++ binaries with a strict defense against vtable attacks.

• Introduces multiple algorithms to reconstruct C++ semantics from binaries without the need for source code, debug symbols, or symbol and type information. VCI employs these algorithms along with inter-procedural type propagation to resolve vcall targets.

• Demonstrates a strict and precise integrity policy that covers all three cases of fully, partially, and unresolved vcall targets. VCI constructs and enforces the policy via static binary rewriting.

• Quantifies the precision of VCI’s policy on various C++ programs, and compares it to the precision of the state-of-the-art binary vtable defenses and to GCC VTV [160]. (GCC VTV is the de facto standard source-based vtable defense of GCC.) Results show that VCI has significantly higher precision than state-of-the-art binary solutions.

• Quantifies the effectiveness of VCI, benchmarks its runtime overhead, discusses how it impacts COOP attacks, demonstrating that VCI can mitigate real-world attacks and incurs a comparable overhead to existing solutions.

8.1 Polymorphism in C++

Commodity applications, such as office suites and web browsers, are built with performance in mind. Given the sophisticated functionalities they provide, it is standard to use languages that provide sufficient levels of abstraction with a minimal performance penalty. Therefore, low-level object-oriented languages, such as C++, are typically the choice for their implementation. To enable polymorphism, C++ uses virtual functions. A function is declared virtual if its behavior (implementation) can be changed by derived classes. The exact function body to be called is determined at runtime depending on the invoking object’s class.

8.1.1 Polymorphism and Virtual Tables

All major C++ compilers, including GCC, Clang/LLVM, MSVC, Linux versions of HP and Intel compilers, use vtables to dispatch virtual functions. A vtable is a reserved read-only table in the binary that
contains function pointers to the definitions of virtual functions accessible through a polymorphic class. A polymorphic class is a class that declares, defines or inherits virtual functions.\(^1\) Each virtual function in a class has a corresponding offset in the class’ vtable which stores the address of the implementation body of the function in the code section. Whenever an object of some class type invokes a virtual function, the class’ vtable is accessed, and the address at the corresponding function offset is loaded and indirectly called. If a class implements virtual functions, when an object of that class type is created, the compiler adds a hidden pointer to the class’ vtable (the vptr). The compiler also generates code in the class’ constructor to set the vptr to the address (effective beginning) of its corresponding vtable.

\(^1\)Unless explicitly stated, I use the term “class” to refer to “polymorphic class” in the rest of this document.
8.1.2 Virtual Call Dispatch

Since a vcall is always invoked on some object, the compiler has to decide how to pass the pointer of the object, i.e., the this pointer, to the callee. There are two widely adopted argument passing conventions for vcalls: thiscall, which is the default convention used by the MSVC compiler on Windows, and stdcall adopted by GCC, LLVM and other Linux compilers. In the thiscall, the this pointer is passed in the ecx register to the callee, while the remaining arguments are passed on the stack. In the stdcall, the this pointer is passed as an implicit argument on the stack (top of stack). The argument is implicit in the sense that it is not part of the callee function signature as seen by the developer.

Figure 8.2 shows the steps taken to dispatch a vcall based on the Itanium ABI, which comprises the following steps:

1. The this pointer of the target object is loaded and dereferenced.
2. An offset is added to the vptr to point to the vtable entry with the address of the target function.
3. The adjusted vptr is dereferenced to load the address of the target function (the vcall address).

Figure 8.2: Assembly snippets for invoking A::bar().

Figure 8.3: Assembly snippet for invoking C::foo() using a base pointer of type B (e.g., B *ptr = new C(); ptr->foo();).

8.1.2 Virtual Call Dispatch

Since a vcall is always invoked on some object, the compiler has to decide how to pass the pointer of the object, i.e., the this pointer, to the callee. There are two widely adopted argument passing conventions for vcalls: thiscall, which is the default convention used by the MSVC compiler on Windows, and stdcall adopted by GCC, LLVM and other Linux compilers. In the thiscall, the this pointer is passed in the ecx register to the callee, while the remaining arguments are passed on the stack. In the stdcall, the this pointer is passed as an implicit argument on the stack (top of stack). The argument is implicit in the sense that it is not part of the callee function signature as seen by the developer.

Figure 8.2 shows the steps taken to dispatch a vcall based on the Itanium ABI, which comprises the following steps:

1. The this pointer of the target object is loaded and dereferenced.
2. An offset is added to the vptr to point to the vtable entry with the address of the target function.
3. The adjusted vptr is dereferenced to load the address of the target function (the vcall address).
4. The this pointer is pushed on the stack.

5. The virtual function is invoked by indirectly calling the vcall address.

Note that step 2 is optional, depending on the index of the target virtual function in the vtable. If it is the first function in the vtable, the offset is 0 and step 2 is omitted. If the virtual function takes arguments, they are all pushed before the this pointer at step 4. While steps 1 – 3 have to occur in that specific order due to data dependency, the ABI does not guarantee the order of steps 1 – 4. For example, pushing the this pointer and the arguments (step 4) can occur before step 1, or even in a different (predecessor) basic block.

In later sections, I use this pattern as part of the algorithm to locate virtual call sites in the binary.

### 8.1.3 Inheritance

C++ supports single, multiple, and virtual inheritance. When a derived class inherits from base classes, the constructor of the derived class calls the constructor of each base class, in the order of inheritance. The derived class passes its this pointer to each base constructor. In the case of multiple inheritance, the this pointer is adjusted to point to the beginning of the base subobject in the derived object’s memory layout. According to the Itanium ABI, inheritance of virtual functions is implemented using multiple vtables, one for each base class. When a derived class C inherits from base classes A and B, an object of type C would contain two subobjects of types A and B, each with its own vtable and vptr. The effective vtable of the derived class consists of a table of vtables (called VTT), one for each subobject type, in the order of inheritance, with the only exception that the derived class and the first subobject share the same vptr. Figure 8.1b illustrates the layout of vtables in memory for single and multiple inheritance.

This leads to the need for this pointer adjustments when using a base pointer to a derived class. For example, Figure 8.3 shows the assembly generated for invoking ptr->foo(), where B *ptr = new C(), i.e., ptr is of base class type besides the first base (first base is A, second is B). The compiler adjusts the pointer before the vcall to point to the subobject B in C (line 2). It then calls (indirectly) a thunk that re-points this to C then jumps to the actual (derived) function body. Similarly, this adjustments are used to access and invoke virtual functions of member class objects (more on this in
Section 8.3.3). VCI keeps track of any adjustments done on identified this pointers and reconstructs the inheritance hierarchy among polymorphic classes.

### 8.2 Problem Definition

Given a C++ program binary, VCI aims to protect the program against vtable attacks by enforcing a strict CFI policy at vcall sites. Specifically, VCI guarantees that for each vcall site, the vcall target is one of the class types that can be legitimately used by that particular vcall site, as statically inferred. If the condition is violated, VCI raises an alarm and terminates the program.

In the following, I discuss my assumptions and give a quick overview of vtable attacks in C++ binaries and how VCI operates.

#### 8.2.1 Assumptions and Threat Model

This work assumes the following: 1. Attackers can read arbitrary readable memory, therefore bypassing any secret-based solution where the secret is stored in readable memory. 2. They can write arbitrary writable memory, including injecting vtables and modifying objects’ layouts and contents. 3. They cannot control the memory protection flags without injecting and executing foreign code. In other words, legitimate control flow transfers in the target program cannot allow the attacker to alter the memory protection of a specific memory region. Those assumptions cover most practical scenarios of attacks, without any unrealistic limits. I also assume that traditional arbitrary code execution defenses are live on the OS, such as ASLR and DEP. This work focuses on vcall protection, which is pertinent to forward-edge control transfers. I assume that other control flow transfers, including non-control-data attacks, are protected and cannot be used to redirect the flow of vcalls.

I assume that the binaries adhere to the Itanium ABI (see Section 8.6); analyzing non-compliant or obfuscated binaries is outside the scope of this work. I assume the binaries are stripped from auxiliary information, such as debug and symbol information, including the C++ RTTI (see Section 8.7), and I do not assume any particular optimization level. While these assumptions complicate the analysis, it is unreasonable to assume the presence of side information when dealing with stripped binaries. The analysis performed in this chapter assumes knowledge of function entry points in the binaries.
I depend on Dyninst [11] in this regard, which has shown outstanding identification accuracy of function entry points in stripped binaries [92, 167], outperforming various well-established static analysis tools.

Finally, various constructs in this chapter require instruction-level analysis that depends on the semantics of the instruction set being parsed. Therefore, I tailor the discussion in this chapter to x86_32 (e.g., call parameters passed on stack instead of in registers as in x86_64). Nevertheless, the approach itself does not put any assumptions on the underlying architecture and can be implemented for other instruction sets without an issue.

8.2.2 Vtable Attacks

By exploiting a memory access bug (e.g., use-after-free [41]), an attacker can launch vtable attacks and achieve arbitrary code execution by overwriting a C++ object’s memory with contents of his or her choice (e.g., via heap spraying [68]). The attacker can inject a fake vtable or perhaps redirect the object’s vptr to an existing vtable. Without loss of generality, vtable attacks in C++ can be divided into three categories [86, 100, 124, 133, 173]:

1. Vtable corruption. This is a legacy attack, where legitimate vtable contents are overwritten. The attack is prevented by all major compilers by storing the vtables in a read-only memory region.

2. Vtable injection. Here, the attacker first injects a fake vtable into memory, then points the vtable pointer of a hijacked object to the injected vtable. The injected vtable can therefore point to arbitrary functions or gadgets in the executable memory of the process.

3. Vtable reuse. This attack operates the same way as a vtable injection attack, except that the attacker does not inject any counterfeit vtables in memory. Instead, the attacker reuses already existing vtables in the process memory.

While the state-of-the-art binary vtable defenses reduce the vtable attack surface, they do not extract sufficient semantics from the binaries, and therefore enforce imprecise policies that allow a very liberal number of target functions per vcall site. To address this limitation, I introduce VCI as a binary rewriting system that fully protects against vtable corruption, injection, and significantly reduces the
attack surface of vtable reuse in C++ binaries. In the following, I give an overview of how VCI operates and give a simplified example of a retrofitted program and the integrity policy enforced by VCI.

8.2.3 Overview of VCI

Figure 8.4 outlines the workflow of VCI. It operates as follows: First, it statically analyzes the binary and extracts all vtables and constructors. It then reconstructs (partially) class layouts and hierarchies. Then, it identifies all vcalls in the binary. VCI then propagates the identified class types to all vcall sites, using backward slicing and inter-procedural data flow analysis. This produces a set of legitimate target class types and their corresponding vtables for each vcall. When VCI fails to resolve all target class types of a vcall, it utilizes the inferred hierarchies and any known targets for the vcall to construct a set of class types that the vcall may be invoked on. As the experiments show, this is significantly more precise than prior works which either liberally permitted any class type to be used at any vcall site, or any class type where the vcall offset is valid. Specifically, VCI constructs and enforces the mapping: $F: vcall \times class \rightarrow vtable$ by instrumenting checks at each vcall site to test if the vcall target class is one of the valid target class types for that vcall.

![Figure 8.4: Overview of VCI. The input to VCI is a binary file (executable or library), and the output is a binary file retrofitted with integrity checks and the VCI integrity enforcement library (libvci).]
8.3 Design and Implementation

8.3.1 Identifying Virtual Tables

To extract vtables, VCI scans the binary for assembly sites that store an address (immediate value) into memory, where the address resides in a read-only memory region, and the words (pointer-size sequence of bytes) at positive offsets of the address are pointers to functions in the code section (see Algorithm 8.1). For each such assembly site, VCI starts with an empty vtable, and scans the corresponding memory region, starting at the stored address, one word at a time. Each word is matched against a set of all function addresses in the binary. If a match is found, the word is added to the vtable, otherwise, the algorithm proceeds on to the next assembly site. According to the Itanium ABI, the vtable address referenced by an object’s vptr (i.e., the entry point of the vtable as seen by the object) is 1. pointer aligned, and 2. points to the beginning of the virtual function pointers array in the vtable. Finally, the algorithm returns the extracted vtable. Note that the algorithm identifies vtables separately. For example, it will identify two separate vtables for class C in Figure 8.1; the VTT of C is populated when VCI reconstructs the class layout.

While VCI may identify false vtables, such as C-style arrays of function pointers (jump tables) stored in a read-only region, the proposed algorithm is sound. It does not miss any real vtable in the binary (no false negatives). This is an important property as missing a legitimate vtable can result in an incorrect policy or misdetection of attacks. Note that C-style jump tables that are misidentified as vtables do not satisfy later stages of the analysis (e.g., no corresponding constructors), and their calling convention does not generally match that of virtual functions (see Sections 8.3.2 and 8.3.4). Overall, overestimation of vtables affects only the precision of VCI rather than its soundness, by increasing the number of potential targets of a vcall.

8.3.2 Identifying Constructors

Each extracted vtable corresponds to one class that declares virtual functions. Each such class will have at least one constructor and one virtual function declaration. To extract constructors, VCI applies Algorithm 8.2. It searches the code section for functions that store a pointer to a vtable at the memory location pointed at by this, i.e., the first entry in an object’s memory. This is done by searching for
Algorithm 8.1: Scan and Extract Vtables

**input**: Rgns: set of memory regions from the binary  
Funcs: set of functions from the binary  

**output**: Vtables: set of virtual function tables

1. foreach `func ∈ Funcs` do
   2. foreach `insn ∈ getInstructions(func)` do
      3. if `writesMemory(insn)` then
         4. `src ← getSrcExpr(insn)`
         5. `dst ← getDstExpr(insn)`
         6. if `isDefined(src)` then
            7. `rgn ← getRegion(src)`
            8. if `readonly(rgn)` then
               9. `vt ← extractVtable(rgn, src)`
      10. `Vtables ← Vtables ∪ vt`
   11. return `Vtables`

Procedure `extractVtable(Funcs, rgn, offset)`

12. `vt ← 0`
13. `i ← 0`
14. foreach `wd ∈ rgn starting at offset` do
15.   if `wd ∈ Funcs` then
       16.     `vt ← vt ∪ {i, wd}`
       17.     `i ← i + 1`
18.   else
       19.     break
20. return `vt`

A function that contains an assembly site that stores an immediate value in memory, where:

1. the immediate value matches the address of one of the extracted vtables;
2. the destination expression has zero displacement; and
3. the destination expression is a memory location pointed at by the first argument to the function, e.g., `mov 0x8(%ebp),%eax; movl $0x9b0,%eax`. Once identified, the vtable is scanned for occurrence of a pointer to that same function. If a pointer to the function is not found in the vtable, the function is deemed a constructor. Note that C++ does not allow virtual constructors, therefore constructors cannot have entries in the vtable. Similarly, inlined constructors are identified by relaxing the first argument condition. In this case, the store instruction that writes the vtable address in the object’s memory is marked as a construction point.
Algorithm 8.2: Identify and Extract Constructors

\begin{algorithm}
\begin{algorithmic}
\State \textbf{input} : \texttt{Funcs}: set of functions from the binary
\hspace{1em} \texttt{Vtables}: set of virtual function tables
\State \textbf{output} : \texttt{Ctors}: set of constructors
\For {\texttt{func} \in \texttt{Funcs}}
\For {\texttt{insn} \in \texttt{getInstructions(func)}}
\State \textbf{if} \texttt{writesMemory(insn)} \textbf{then}
\State \texttt{src} \leftarrow \texttt{getSrcExpr(insn)}
\State \texttt{dst} \leftarrow \texttt{getDstExpr(insn)}
\State \textbf{if} \texttt{isDefined(src)} \textbf{and} \texttt{getDisp(dst)} = 0 \textbf{and} \texttt{firstArg(dst)} \textbf{then}
\State \texttt{vt} \leftarrow \texttt{Vtables[src]}
\State \textbf{if} \texttt{vt} \neq \emptyset \textbf{then}
\State \textbf{if} \texttt{getOffset(func)} \notin \texttt{vt} \textbf{then}
\State \hspace{1em} // ctor cannot be in vt
\State \texttt{Ctors} \leftarrow \texttt{Ctors} \cup \texttt{func}
\EndWhile
\EndFor
\EndWhile
\end{algorithmic}
\end{algorithm}

8.3.3 Inferring Class Layouts and Hierarchies

When a derived class inherits from a base class, the constructor of the derived class calls the base constructor, passing in the \texttt{this} pointer of the derived class after applying any necessary pointer adjustments (in case of multiple inheritance). The same semantics are also applied when constructing member objects.

VCI infers class layouts that consist of offsets to polymorphic member objects and base subobjects, and offsets to vtables (the VTT in case of multiple inheritance). The offsets are computed relative to the class \texttt{this} ptr. I collectively refer to member objects and base subobjects as subelements. Each subelement is defined by the tuple: \langle \texttt{cls}, \texttt{offset}, \texttt{dst}, \texttt{deref} \rangle, where \texttt{cls} is the containing class, \texttt{offset} is the subelement's offset from the \texttt{this} pointer of \texttt{cls}, and \texttt{dst} is the corresponding subelement's class. \texttt{deref} is a flag indicating whether the subelement has to be dereferenced before accessing, e.g., if a member is a pointer to an object, where in this case the class stores only the subelement's \texttt{this} pointer instead of the subelement itself.

Algorithm 8.3 outlines the steps taken to infer the layout. For each class, VCI infers the class layout by, first, searching the instructions of the class constructor for assembly call sites that invoke a constructor. Then, for each identified call site, it extracts and analyzes the arguments to the call.
site to identify the this pointer of the subelement’s constructor. It then computes the offset of the subelement’s this pointer to the class this pointer. Recall that the this pointer points to the address at which the vtable pointer is stored in an object’s memory. VCI computes the offset by analyzing the adjustments performed on the this pointer before calling the subelement’s constructor. For example, mov 0x8(%ebp),%eax; add 0x4,%eax; mov %eax,(%esp); call sub_ctor(); constructs a subelement at offset 0x4 from the this pointer of the class. This results in an expression of the form this + offset, where this is the class this pointer, and offset is the distance to the subelement’s this pointer. Finally, VCI checks if the subelement needs to be dereferenced before accessing by checking if the this pointer passed to the subelement’s constructor is stored in memory after the call to the subelement’s constructor.

### Algorithm 8.3: Reconstruct Class Layout

**input**: cls: initial class layout  
Ctors: set of constructors  
**output**: cls: populated class layout

1. offset ← 0  
2. foreach insn ∈ getInstructions(cls.ctor) do  
3. if isCall(insn) then  
4.   dst ← getCallTarget(insn)  
5.   if dst ∈ Ctors then  
6.     mThis ← findThis(dst)  
7.     offset ← calcOffset(cls.this, mThis)  
8.     deref ← storesThis.cls, mThis)  
9.     addToLayout(cls, offset, dst, deref)  
10. return cls

Similarly, VCI populates the class VTT by identifying the assembly site that stores pointers to vtables, relative to the this pointer of the class. For example, mov 0x8($ebp),%eax; mov $0x848,(%eax); add $0x8,%eax; mov $0x88c,(%eax); corresponds to a VTT of two entries 0x848 at offset 0 and 0x88c at offset 0x8. Note that the first entry of the VTT is the class’ vtable itself.
To reconstruct inheritance relationships between polymorphic classes, VCI needs to differentiate between calls to a base constructor and calls to construct member objects. According to the ABI, in a derived class, its virtual base class’ subobjects are constructed before its member objects. In addition, the compiler has to populate the VTT of the class before constructing its member objects. In other words, all calls to constructors that 1. take the derived class’ this pointer (adjusted) as the top argument on the stack, and 2. occur before storing the vtable address at a zero offset from the this pointer in the object’s memory, are calls to base constructors. By identifying this pattern in the assembly of constructors, VCI constructs the “is-a” relationship among the identified polymorphic classes. Note that the actual offsets in the VTT in the binary must match the offsets VCI extracted for inherited classes. Additionally, the soundness of the inferred hierarchy follows from the soundness of VCI’s vtable identification (no FNs). VCI does not attempt to construct the full class hierarchy that includes polymorphic and non-polymorphic classes, which is a known hard problem [81, 101, 119]. VCI uses the identified inheritance hierarchy to augment its policy when semantic gaps hinder the identification of all class types that a vcall operates on (see Section 8.3.6).

8.3.4 Identifying Virtual Calls

To extract call sites that invoke virtual functions, i.e., vcall sites, VCI scans the binary for indirect call sites that reflect the behavior of virtual function dispatches. That is, the indirect call target is computed by first dereferencing a pointer (the vtable pointer), then adjusting the resulting address to pick an entry of the vtable by adding a non-negative constant offset to it, and finally dereferencing the final adjusted address to retrieve the address of the target function. In addition, the same expression used to dereference the vtable (the this pointer) is passed as the first argument (top of stack) to the indirect call. Note that the offset used in a vcall site is not a target of attacks as it is always hardcoded in the assembly as an immediate value or a displacement. However, attackers can effectively change the offset by modifying the vtable address (vptr) referenced by the this pointer.

Though a vtable and a jump table (an array of function pointers) share common structure, the semantics for invoking virtual functions are different from those for dispatching functions from a jump table. To dispatch a function pointer from a jump table, the jump table is directly indexed rather than offset and dereferenced. For example, given an index in %ecx and a jump table stored
at address 0xa034, the target function from the jump table is invoked by: call 0xa034(,%ecx,4).\(^2\)

This dissimilarity in how the indirect call target is computed enables VCI to filter out any spurious jump tables that might have been mislabeled as vtables by Algorithm 8.1.

This approach in itself does not yield FPs (false positives, i.e., incorrectly identifying a call as a vcall) for ABI-compliant binaries. However, it might incorrectly identify some specific C constructs as vcalls in mixed C/C++ binaries. Besides that, special compiler rearrangements and nonstandard calling conventions that may not be handled by the implementation can result in FNs (false negatives, i.e., missing valid vcalls). I evaluate the identification accuracy of VCI in Section 8.4.1 and discuss vcall-like C constructs in Section 8.7.3.

8.3.5 Class Type Propagation and Pairing

Intra-Procedural Analysis

VCI implements a custom, slicing-based, intra-procedural analysis algorithm (illustrated in Algorithm 8.4) to construct intra-procedural bindings between classes and (v)calls. It starts by analyzing the assembly sites at which calls are invoked. For each call site, it extracts a backward slice, starting at the assembly store point of each argument to the call site and ending at the entry point to the procedure. VCI analyzes the slice and decides whether and what classes the argument depends on, i.e., there is data dependence between the call parameter and one or more this pointers defined within the same procedure.

Each backward slice is a Program Dependency Graph (PDG) constructed via Value-Set Analysis (VSA) [47]. Nodes in the PDG correspond to program constructs (assignment expressions) and edges correspond to data and control dependencies between the assignments. Since there is no notion of variables at the binary level, VSA extracts variable-like abstractions using the semantics of the instructions. Due to the multi-assignment (multi-source, multi-destination) nature of assembly instructions, the produced slices are often cluttered with irrelevant expressions and dependency paths [157]. To overcome this, VCI analyzes the slice by traversing backwards all paths in the PDG from the exit node (i.e., the call argument) to each entry node. For each path, VCI traces (backwards)

\(^2\)Code generated by both GCC and Clang with -O1, -O2, and -O3. For -O0, GCC and Clang emitted: mov %ecx,%eax; mov 0xa034(,%eax,4),%eax; call *%eax, which also does not satisfy the semantics of a vcall.
Algorithm 8.4: Intra-procedural Type-Vcall Pairing

**input**: Ctors: set of constructors
Funcs: set of functions from the binary

**result**: Pairing information between classes and vcalls

```
1 foreach func ∈ Funcs do
2     foreach call ∈ func do
3         foreach param ∈ findParams(call) do
4             slice ← backwardSlice(param)
5                 foreach entryNode ∈ slice do
6                     def, Adjs ← reaches(entryNode, exitNode)
7                     if def ≠ ∅ then
8                         cls ← resolve(def, Adjs, func, Ctors)
9                         pair(cls, param, call, Adjs)
10                    else
11                       pair(∅, param, call, Adjs)
```

the data flow of the argument through memory and registers, till it reaches a construction site (the defining constructor) or an entry node. During this, VCI also maintains a list of all encountered adjustments (via offsets and displacements) that were performed on the parameter expression, in their order of execution. See Figure 8.6 for an example snippet and its corresponding PDG generated by VCI.

If such flow exists, then a data dependency is present between the call parameter and that construction site (and its corresponding class). In this case the parameter type is resolved by pairing it with the effective class resulting from the corresponding construction site class after applying any this pointer adjustments. If the resolved parameter is the first parameter of the vcall (i.e., the this pointer), VCI resolves the vcall target using the resolved parameter’s vtable and the vcall offset. It then adds an edge to the CFG between the vcall and the resolved virtual function address.

If there is no data flow, then the path is ignored. Finally, if a flow exists but the defining constructor was not found, that could mean either the definition point is in a different procedure or there is a semantics gap, which I discuss in the following sections.
Inter-Procedural Analysis

VCI performs inter-procedural analysis by recursively propagating the this definitions and parameter type information of each procedure down the CFG, through returns and successor call sites. The analysis traces the this pointers of both class objects and members as identified in Section 8.3.3. Specifically, class types are propagated across function boundaries by checking the equivalence of the expressions of the arguments pushed on the stack at the call site (in the caller function) with those loaded from the stack in the preamble of the callee. For example, %edx and 0x8(%ebp) in the following snippet are equivalent: foo: push %edx; call bar; and bar: mov 0x8(%ebp),%eax.

For returns, class types are recursively propagated across all procedures exit points (function returns) if there is a data dependency between the exit point and the this pointers of the objects referenced in the function body.

Vcall Target Resolution  For vcalls, VCI attempts to resolve the vcall target by identifying data flows from the incoming this pointer expressions on the stack of the enclosing (parent) procedure, to the arguments of the vcall. Similar to Section 8.3.5, this is done by pairing arguments to incoming (adjusted) class types via reachability analysis over a backward slice starting at each argument to the vcall and ending at the entry of the enclosing procedure. If the first argument (i.e., the this pointer the vcall is invoked on) type is successfully resolved, VCI finds the corresponding virtual function address in the corresponding class' vtable, at the offset that appears in the vcall site, and adds an edge to the CFG between the vcall and the virtual function address. The algorithm stops when the CFG stops changing.

Due to semantic gaps (see Section 8.5), it is possible that VCI fails to resolve all definition points of a vcall's this pointer, resulting in potentially missing some valid vcall targets. This divides vcalls into three categories: 1) fully resolved vcalls, where all definition points were successfully paired; 2) partially resolved vcalls, where some but not all definition points were resolved; and 3) unresolved vcalls where all definition points were not paired with any type. In the following, I discuss how VCI generates and enforces its policy such that it covers all the three cases, yet be as strict as possible.
8.3.6 Policy Generation and Enforcement

VCI generates the following policy, based on vcall target resolution results:

1. For fully resolved vcalls, all legit targets were successfully identified, and only those targets are considered valid.

2. For partially resolved vcalls, find the common base classes among the identified targets and all child classes that inherit from those common bases (including the identified targets). Assume that all vtable functions at the vcall offset in those classes are valid targets.

3. For unresolved vcalls, i.e., no targets were identified for the vcall, assume that all vtable functions at the same vcall offset are valid targets (the same integrity policy applied by Prakash et al. [133]).

The policy is implemented by constructing the mapping $F: vcall \times class \rightarrow vtable$. VCI stores in the binary a read-only set $C$ of the extracted class types and their inferred layouts, one class for each nonempty vtable. Each class is assigned a unique ID. Then, before each vcall site $v$, VCI aggregates a set of IDs $L_v$ of all the class types that $v$ can be invoked on, based on the three policy cases (for fully, partially, and unresolved vcalls). It then injects code in the binary that enforces $F$ by checking for the following:

1. There is a class $c$ in $C$ with the same vtable address of the class $c'$ accessed by the vcall.

2. The ID of $c$ belongs to the valid IDs $L_v$, i.e., $c_{ID} \in L_v$.

3. The class layout of $c'$ matches the layout of $c$.

If any of the conditions is not met, the execution is aborted and an alarm is raised. Otherwise, the vcall is dispatched. No policy is enforced for static (direct) call sites (e.g., $call \_ \_ 660$) since they do not pose a threat under VCI's threat model. Checking the class layout is important in order to detect reuse attacks that modify the vptr of an object to point to a different vtable than the actual object's vtable, where both vtables are valid for the vcall site. The layout checks validate the information extracted in Section 8.3.3 (the contents of all involved vtables and offsets of subobjects from each this pointer).
In my prototype implementation, the definitions of the policy enforcement procedures are exported in a dynamic library (libvci) that VCI injects into the binary. At runtime, libvci linearly checks the policy conditions on the valid classes set of \( v \), i.e., \(|C [L_v[i]] | \ i \in 1 \ldots |L_v|\), where \(|L_v| \leq |C|\). A potential performance improvement is to add a sublinear index, such as using binary search over vtable addresses whenever \( \log |C| < |L_v| \), or a read-only hash table that maps vtable addresses to classes. I decided to go with linear constant arrays for simplicity and to avoid unintentionally introducing writable memory or more attack points.

I conclude this section by providing a complete example in Figure 8.5. The figure shows an example C++ program, its corresponding assembly dump, and the policy semantics injected by VCI at the vcall site. The corresponding filtered PDG generated and analyzed by VCI is shown in Figure 8.6.

8.4 Evaluation

In my evaluation of VCI, I answer the following questions:

1. How accurately can VCI identify vtables and vcalls? Section 8.4.1.

2. How precise and effective is the policy enforced by VCI, compared to both binary and source-based state-of-the-art C++ defenses? Section 8.4.2.

3. How much runtime overhead do binaries protected by VCI incur? Section 8.4.3.

All experiments were conducted with GCC 4.8.2 on Ubuntu 14.04.1, running on 2.5GHz Intel Core i7 with 16GB RAM. The results are reported for -m32 and -O2 optimization, but similar results were observed at other optimization levels.

8.4.1 Identification Accuracy

I compiled the C++ SPEC CPU2006 benchmarks and the C++ Firefox modules\(^3\) with debug and symbol information, then counted the number of nonempty vtables by parsing the output of the `objdump -Ct` command, which demangles and dumps the symbol table entries of a binary. That count is used as the ground truth. I then compiled the same programs without debug and symbol information,

\(^3\)For the non-C++ FireFox modules, VCI did not identify any vtables in the binaries and aborted the analysis without modification to the binaries.
int main() {
    int x;
    cin >> x;
    Base *ptr = nullptr;
    if (x == 1) ptr = new A();
    else ptr = new B();
    ptr->foo(); // vcall
}

(a) Example C++ program with a virtual call.

(b) Assembly dump of a

(c) Injected policy checks before the vcall at 7fd.

Figure 8.5: (a) Example C++ program, (b) it’s assembly dump, and (c) the policy injected by VCI. clsz is the statically constructed set of valid classes at the vcall site. ptr refers to the pointer at address 0x1c(%esp).
processed them by VCI, counted the number of extracted vtables and compared to the ground truth. Here, FNs (missing a vtable) are not desired, while FPs are acceptable since the policy is enforced at vcall sites rather than the vtables themselves. In other words, falsely identified vtables will not result in FPs at runtime, but in lower precision during the identification of vcall targets.

I report the count of vcalls in each binary, and compare that to the ground truth from GCC VTV. VTV inserts checks at each vcall site in the binary to validate its vtable. I compiled each of the test programs with and without VTV, and matched the call sites that contained VTV checks against the vcall sites identified by VCI. Note that, unlike vtables, falsely identified vcalls may result in runtime crashes. Thus, FPs in terms of vcalls are undesired, or else the enforced policy would be unsound. On the other hand, FNs (missed vcalls) do not sway the soundness of the policy rather reduce its precision.

Table 8.1 shows the breakdown of the analysis results. VCI did not miss any legitimate vtable in the binaries, achieving zero FNs. It incorrectly identified some memory blocks as vtables in five out of the 13 binaries, resulting in FPs between 0.04% and 3.33%. In terms of vcalls, VCI did not report any FPs, but it had some FNs (missed vcalls) between 0.32% and 2.18%. These results indicate that VCI shall be sound, but not perfectly precise (not complete) due to the missed vcalls and the overestimated vtables. I quantify the precision of VCI in the following section.
Table 8.1: Analysis result of the C++ SPEC CPU2006 benchmarks (top) and the C++ Firefox modules (bottom), including the analysis time in seconds, number of identified vtables and vcalls, and the identification accuracy.

<table>
<thead>
<tr>
<th>Program</th>
<th>#Insns</th>
<th>Analysis Time (sec.)</th>
<th>Identified Vtables</th>
<th>Identified Vcalls</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Ground Truth</td>
<td>VCI</td>
</tr>
<tr>
<td>444.namd</td>
<td>91k</td>
<td>22.8</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>447.dealII</td>
<td>789k</td>
<td>193.2</td>
<td>732</td>
<td>736</td>
</tr>
<tr>
<td>450.soplex</td>
<td>110k</td>
<td>128.4</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>453.povray</td>
<td>256k</td>
<td>143.4</td>
<td>30</td>
<td>31</td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>166k</td>
<td>207.5</td>
<td>114</td>
<td>114</td>
</tr>
<tr>
<td>473.astar*</td>
<td>12k</td>
<td>0.2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>483.xalanchmk</td>
<td>1m</td>
<td>229.6</td>
<td>962</td>
<td>971</td>
</tr>
<tr>
<td>libgblib.so</td>
<td>9k</td>
<td>12.3</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>libmozgname.so</td>
<td>53k</td>
<td>134.7</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>libmozjs.so</td>
<td>2m</td>
<td>510.4</td>
<td>1230</td>
<td>1244</td>
</tr>
<tr>
<td>libxul.so</td>
<td>27m</td>
<td>2973.6</td>
<td>14465</td>
<td>14471</td>
</tr>
<tr>
<td>libzmq.so</td>
<td>122k</td>
<td>108.2</td>
<td>67</td>
<td>67</td>
</tr>
<tr>
<td>updater</td>
<td>31k</td>
<td>15.3</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

* I manually inspected 473.astar and found that it contained 4 indirect calls, none of which were vcalls.

8.4.2 Security Effectiveness

Policy Precision

Quantifying the effectiveness of a defense system is a difficult task. Recent work by Zhang et al. [176] introduced the Average Indirect-Target Reduction (AIR) metric as a quantitative measure of the security introduced by a defense. I understand that the AIR metric has been questioned by the community [57], primarily since it does not quantify the usefulness of the remaining targets from the attacker’s perspective. However, for the sake of comparison with similar defenses, I use an AIR-based metric as part of my evaluation. I concede that a better evaluation metric is needed, albeit outside the scope of this work. Developing a conclusive security metric is a very challenging task, especially when dealing with whole functions as in the case of VCI. To give conclusive results, I also compare the precision of VCI to that of GCC VTV, the state-of-the-art source-based vtable defense.

In the context of VCI, we are only interested in defending vcalls, which are forward-edge control transfers. Therefore, I only compute the average number of vcall targets over all vcalls. I protected the C++ SPEC CPU2006 benchmarks and the C++ Firefox modules, and computed the average number of targets per vcall. Then, I computed the precision of the policy as the percent reduction in the average number of vcall targets, compared to the source-based defense GCC VTV (perceived as the ground truth) as well as the two policies that appeared in prior studies.
1. “AnyV,” permit the vcall target to be any function in any vtable (e.g., [124, 173]); and

2. “SameOff,” permit only functions at the same vtable offset as the vcall site, in any vtable (e.g., [86, 133]).

The higher the reduction the more precise the enforced policy. I further assumed that the solutions that enforced any of those two policies had perfect knowledge of the vtables in the binaries. Since exploits do not only target vcalls, and for the sake of completeness, I also report the reduction in attack surface on indirect calls (icalls). This is computed as the percentage of vcalls (protected by VCI) to the total number of icalls in the analyzed binaries.

Table 8.2 shows the breakdown of the results per program. The results show that VCI achieved significantly higher precision that prior solutions. For some programs, it limited the vcall target to one or two functions on average (e.g., 444.namd, Firefox liblgllibs.so and updater). In comparison to the source-based VTV, VCI achieved the highest precision amongst other policies, with 100% precision in some cases, and greater than 60% precision for the majority of the programs. Compared to solutions that apply the AnyV policy, VCI achieved 87% to 99% reduction in the vcall targets. This is more pronounced in programs with large numbers of vcalls. For example, in Firefox libxul.so, VCI limited each vcall to only 1035 targets on average, while AnyV allowed 71069 targets per vcall. Compared to SameOff policies, VCI achieved 48% to 89% reduction. For the same libxul.so, a SameOff policy would permit 9692 targets per vcall, while VCI reduced that by more than 89%.

**Real-World Exploits**

I experimented with three publicly-available use-after-free vtable exploits for Mozilla Firefox: CVE-2011-0065, CVE-2013-0753, and CVE-2013-1690. All three vulnerabilities reside in libxul. CVE-2011-0065 exploits a use-after-free vulnerability in Firefox 3.6.16 where the mChannel pointer associated with an Element object can be used after being freed, via the OnChannelRedirect function of the nsIChannelEventSink class. CVE-2013-0753 exploits a vulnerability in Firefox versions prior to 17.0.2, where an object of type Element is used after being freed inside the the serializeToStream function of the nsDocumentEncoder class. CVE-2013-1690 exploits a vulnerability in Firefox 17.0.6 where a DocumentViewerImpl object is used after being freed, when triggered via a specially crafted web page.
Table 8.2: VCI policy coverage and average target reduction in the analyzed programs. AnyV refers to solutions that allow any target as long as it is in a valid (read-only) vtable. SameOff refers to solutions that allow targets that are in a valid vtable and at the same offset of the vcall site. VTV represents the source-based ground truth.

<table>
<thead>
<tr>
<th>Program</th>
<th>Avg. #targets per vcall</th>
<th>%Precision w.r.t VTV</th>
<th>%Reduction vs.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AnyV</td>
<td>SameOff</td>
<td>VCI</td>
</tr>
<tr>
<td>444.namd</td>
<td>8</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>447.dealII</td>
<td>3775</td>
<td>289</td>
<td>49</td>
</tr>
<tr>
<td>450.soplex</td>
<td>725</td>
<td>17</td>
<td>8</td>
</tr>
<tr>
<td>453.povray</td>
<td>384</td>
<td>41</td>
<td>13</td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>2361</td>
<td>72</td>
<td>37</td>
</tr>
<tr>
<td>483.xalancbmk</td>
<td>11345</td>
<td>515</td>
<td>85</td>
</tr>
<tr>
<td>libglib.so</td>
<td>61</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>libmozgnome.so</td>
<td>149</td>
<td>20</td>
<td>9</td>
</tr>
<tr>
<td>libmozjs.so</td>
<td>23840</td>
<td>657</td>
<td>91</td>
</tr>
<tr>
<td>libxul.so</td>
<td>71069</td>
<td>9692</td>
<td>1035</td>
</tr>
<tr>
<td>libzmq.so</td>
<td>979</td>
<td>42</td>
<td>13</td>
</tr>
<tr>
<td>updater</td>
<td>44</td>
<td>7</td>
<td>2</td>
</tr>
</tbody>
</table>

using the onReadyStateChange event and the Window.stop API. This was the vulnerability exploited in the wild in 2013 to target Tor Browser users.

I downloaded vulnerable Firefox versions, protected the relevant C++ modules with VCI and tested the protected browser against exploits from Metasploit. Though some Metasploit modules for the aforementioned vulnerabilities supported only Windows, the HTML payloads that trigger the vulnerabilities are cross-platform. The only platform-specific part is the actual payload (ROP in all 3 exploits) that is executed after the vulnerability is exploited.

VCI identified and protected the vcalls targeted by the exploits, rendering the three exploits inoperable. All three exploits resembled a vtable injection attack. I could not find any publicly-available vtable reuse attacks. In the following, I discuss how VCI mitigates and hardens binaries against COOP attacks.

Impact of VCI on COOP

Schuster et al. [65,148] introduced Counterfeit Object-Oriented Programming (COOP), a novel vtable reuse attack against C++ programs. In a COOP attack, the attacker injects a counterfeit (attacker controlled) object that repurposes existing virtual functions in the binary. The counterfeit object is specially crafted such that benign vulnerable constructs in the binary execute attacker picked virtual
functions. The gadgets in a COOP attack are calls to virtual functions (vfgadgets). By chaining multiple vfgadgets via counterfeit objects, the attacker can achieve arbitrary code execution.

A COOP attack requires a memory corruption bug that enables injection of attacker-controlled objects. Besides that, it has two key requirements for a successful exploit: 1. the ability to target unrelated virtual functions from the same vcall site; and 2. the ability to flow data between the vfgadgets. The vfgadgets are dispatched via two types of initial vfgadgets: the main-loop gadget (ML-G), and the recursive gadget (REC-G). The ML-G gadget represents a linear dispatch using a loop that iterates over a list of objects (counterfeit) and calls some virtual function of each object. The REC-G gadget corresponds to a recursive dispatch using two consecutive vcalls on different objects, where the first vcall dispatches one vfgadget and the second vcall recurses back into a REC-G.

In a COOP attack, data is passed between vfgadgets either explicitly or implicitly. In explicit data flows, the attacker picks vfgadgets that pass data via object fields or vcall arguments. In implicit data flows, data is passed via unused argument registers by chaining vfgadgets that take different numbers of arguments. Note that this is specific to architectures that pass arguments in registers by default, such as x86_64. Explicit data flow via object fields is achieved by overlapping objects, in memory, of different classes such that one vfgadget writes to some object field, then another vfgadget reads from the same field. On x86_32, this also requires an initial vfgadget that passes the same field to the dispatched vfgadgets, so that they can read or write to it. In the following, I discuss how VCI abrogates the attacker’s ability to satisfy the COOP requirements.

Tables 8.3 and 8.4 provide summary statistics of VCI’s vcall target resolution results. The statistics represent the number of vcall targets per vcall site for each of the three policy cases. VCI fully and partially resolved 58% plus 26% of all vcall targets, on average (geometric). Unresolved targets ranged from 0% to 29%, with an average of 14%. While the percentage of unresolved calls is not particularly low for some of the test programs, the percentage of fully and partially resolved targets outweighed that of unresolved targets in all programs.

For both fully and partially resolved vcalls, VCI guarantees that all targets of a vcall are at the same vtable offset and under the same class hierarchy. The targets in this case correspond to function polymorphs (redefinitions) of some virtual function in the hierarchy, therefore, all taking the same number of arguments. This prevents implicit data flows in COOP. This also means that the targets
Table 8.3: Vcall target resolution statistics.

<table>
<thead>
<tr>
<th>Program</th>
<th>Fully</th>
<th>Partially</th>
<th>Unres.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min.</td>
<td>Max.</td>
<td>Avg.</td>
</tr>
<tr>
<td>444.namd</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>447.deal</td>
<td>1</td>
<td>31</td>
<td>18</td>
</tr>
<tr>
<td>450.soplex</td>
<td>1</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>453.povray</td>
<td>1</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>1</td>
<td>28</td>
<td>19</td>
</tr>
<tr>
<td>483.xalanck</td>
<td>1</td>
<td>49</td>
<td>31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program</th>
<th>Min.</th>
<th>Max.</th>
<th>Avg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>libgblibss.so</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>libmozgnome.so</td>
<td>1</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>libmozjs.so</td>
<td>1</td>
<td>62</td>
<td>33</td>
</tr>
<tr>
<td>libxul.so</td>
<td>1</td>
<td>208</td>
<td>74</td>
</tr>
<tr>
<td>libzmq.so</td>
<td>1</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>updater</td>
<td>1</td>
<td>11</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 8.4: Percentage of fully, partially, and unresolved vcalls of the C++ SPEC CPU2006 benchmarks (top) and the C++ Firefox modules (bottom).

<table>
<thead>
<tr>
<th>Program</th>
<th>Identified Vcalls</th>
<th>%Resolved Vcalls</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fully</td>
<td>Partially</td>
</tr>
<tr>
<td>444.namd</td>
<td>1</td>
<td>100%</td>
</tr>
<tr>
<td>447.deal</td>
<td>916</td>
<td>37%</td>
</tr>
<tr>
<td>450.soplex</td>
<td>511</td>
<td>63%</td>
</tr>
<tr>
<td>453.povray</td>
<td>127</td>
<td>51%</td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>706</td>
<td>47%</td>
</tr>
<tr>
<td>483.xalanck</td>
<td>9134</td>
<td>56%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program</th>
<th>Identified Vcalls</th>
<th>%Resolved Vcalls</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fully</td>
<td>Partially</td>
</tr>
<tr>
<td>libgblibss.so</td>
<td>63</td>
<td>79%</td>
</tr>
<tr>
<td>libmozgnome.so</td>
<td>206</td>
<td>72%</td>
</tr>
<tr>
<td>libmozjs.so</td>
<td>3784</td>
<td>60%</td>
</tr>
<tr>
<td>libxul.so</td>
<td>79315</td>
<td>32%</td>
</tr>
<tr>
<td>libzmq.so</td>
<td>133</td>
<td>32%</td>
</tr>
<tr>
<td>updater</td>
<td>8</td>
<td>75%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program</th>
<th>Identified Vcalls</th>
<th>%Resolved Vcalls</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fully</td>
<td>Partially</td>
</tr>
<tr>
<td>geomean:</td>
<td>58%</td>
<td>20%</td>
</tr>
</tbody>
</table>

are functionally related, since they are part of the same hierarchy, and unlikely to exhibit useful semantics for the attacker.

For explicit data flows, VCI checks the layout of a vcall invoking object and all its polymorphic subobjects against the statically inferred layouts. The checks assert that the contents of all involved vtables and offsets of subobjects from each this pointer are valid. This means that the attackers cannot overlap objects unless the objects classes have a sufficient number of non-polymorphic subobjects arranged in a way that allows overlapping without disrupting the layout checks. This arguably significantly complicates the attacks. Additionally, even if such classes were available, VCI guarantees the integrity of vtable contents which consequently prevents the invocation of desired
system calls via counterfeit vtables. This limits the attackers to only invoking system calls via vfgadgets that invoke an attacker-controlled indirect call (C-style function pointer), which are “rare in practice” [148] and outside the scope of this work.

If VCI fails to resolve the targets of some vcall (14% of all vcalls in Table 8.4, on average), it resorts to the SameOff policy for that particular vcall. This might enable an attacker to deploy a COOP attack by attacking and utilizing only the unresolved vcalls. Note that the same policy against counterfeit object overlapping is still in effect for unresolved vcalls. While attackers might be able to workaround those constraints, at least in theory, this setup is still significantly constrained compared to unprotected binaries. The reduction in attack surface is essential to heighten the cost of building a functional exploit. Complementary solutions that depend on reference and argument counts (e.g., [165, 175, 176]) can be selectively applied at unresolved vcalls sites to further shrink the possibility of data flows (see Section 8.5).

8.4.3 Performance Overhead

I benchmarked the runtime overhead of binaries protected by VCI using 1) the C++ SPEC CPU2006 benchmarks, and 2) the three industry standard browser speed benchmarks: JetStream, Kraken and Octane. The results are tabulated in Table 8.5. Overall, VCI incurred low overhead ranging from 2.01% to 10.69% on namd, dealII, soplex, and povray. omnetpp and xalancbmk incurred higher overhead (21.11% and 34.80%), which is believed to be a side effect of alignment changes in the modified binary [86, 100, 160]. On browser benchmarks, VCI incurred very low overhead, ranging from about two to seven percent. Overall, VCI incurred a total average (geometric) of 7.79%. The time it took VCI to analyze each binary is tabulated separately in Table 8.1. The overhead incurred by VCI aligns with the state-of-the-art vtable defenses (10% – 18.7% [133], 0.6% – 103% [86], 2% – 30% [100], 8% – 19.2% [160]).
Table 8.5: Performance overhead of VCI on the C++ SPEC CPU2006 benchmarks and three industry standard browser speed benchmarks on Firefox (median of 3 runs).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>orig. s</th>
<th>new s</th>
<th>overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>444.namd</td>
<td>739</td>
<td>818</td>
<td>10.69%</td>
</tr>
<tr>
<td>447.dealII</td>
<td>1813</td>
<td>1994</td>
<td>9.98%</td>
</tr>
<tr>
<td>450.soplex</td>
<td>565</td>
<td>613</td>
<td>8.50%</td>
</tr>
<tr>
<td>453.povray</td>
<td>399</td>
<td>407</td>
<td>2.01%</td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>612</td>
<td>825</td>
<td>34.80%</td>
</tr>
<tr>
<td>483.xalancbmk</td>
<td>1047</td>
<td>1268</td>
<td>21.11%</td>
</tr>
</tbody>
</table>

JetStream: 146.64 pt 135.81 pt 7.34%
Kraken: 1332.7 ms 1358.5 ms 1.94%
Octance: 27328 pt 25819 pt 5.52%

geomean: 7.79%

8.5 Discussion and Improvements

In this section, I discuss more technical aspects, limitations, and improvements of VCI, and compare VCI to related policies.

8.6 ABI Dependency

The C++ Application Binary Interface (ABI) sets the interface between program modules and the execution environment at the assembly level. It defines things such as the memory layout of objects, details of how virtual functions are invoked, and the behavior of the linking stage. The most adopted C++ ABI is the Itanium ABI [98], which is the focus of this work. The Itanium ABI is used by all Linux compilers. Alternatively, the MSVC compiler on Windows uses the MSVC ABI which was internally developed by Microsoft. The two ABIs mainly differ in their choice of calling conventions and the layouts of vtables in memory. Nevertheless, the approach discussed in this chapter can also be applied to MSVC C++ binaries by adjusting the algorithms to accommodate the rules of MSVC.

8.7 Why not depend on RTTI?

C++ supports dynamic type reporting, i.e., identifying and checking the actual type of an object at runtime (as opposed to at compile-time). This is enabled by what the ABI calls “Runtime Type Information” (RTTI). RTTI enables the program to dynamically identify and cast objects at runtime, via
the typeid and the dynamic_cast operators, respectively. For each polymorphic class, an RTTI record is added to the class layout in memory, and a pointer to that record is included at a negative offset in the vtable. The RTTI record contains several structures that describe the class type and its bases.

The structural details of RTTI records can be very useful in reconstructing the polymorphic class hierarchy. However, RTTI is not required if the program uses an RTTI operator in a way that the compiler can infer at compile-time. For example, a dynamic up cast to an unambiguous base can be replaced by a static (compile-time) cast by the compiler, hence not requiring RTTI. All major compilers support RTTI as an optional feature that can be enabled or disabled. Some compilers, such as Clang/LLVM, use alternative implementations of RTTI via C++ templates (dyn_cast<>, isa<> in Clang). Additionally, RTTI is typically stripped from COTS binaries or is not present to begin with. For instance, the C++ Firefox modules on Ubuntu that were used in the experiments had no RTTI by default. Finally, the details of the RTTI record are compiler-specific, and the ABI does not mandate an implementation standard for compilers to follow. Therefore, I opted against depending on RTTI in VCI.

8.7.1 Position-Independent Code (PIC)

VCI supports position-independent code (PIC), including executable and shared libraries. For instance, the Firefox modules used in the experiments were all PIC. To support PIC, VCI first analyzes the binary by searching for a memory section with a data.rel prefix, which is the prefix used to denote relocatable data regions in binaries. If any such section is identified, VCI extracts all program counter thunks (PC thunks) in the binary. A PC thunk is a function generated by the compiler to load the current PC into a specific register when called, which allows memory accesses as an offset from the PC. VCI identifies PC thunks by searching for two-instruction functions that move the stack pointer to a register and immediately return, e.g., the function get_pc_thunk.cx: mov (%esp),ecx; ret; returns the PC into the ecx register when called. Recall that the call instruction pushes the address of the immediately proceeding instruction on the stack, and global data is accessed via an offset relative to the PC in PIC. Once PC thunks are identified, the analysis proceeds as normal, with the only exception that the PC value returned by PC thunks, and the PC offset, are taken into consideration when computing vtable addresses during the extraction of vtables and constructors.
8.7.2 Heterogeneous Containers

VCI, like any static analysis solution, has limited visibility into the semantics of the analyzed programs. Despite that VCI extracts significantly more semantics than prior solutions, there are cases where the analysis fails to identify all the class types used by a vcall. The most common case is objects stored in a heterogeneous container, e.g., a container of base pointers. Even though VCI performs alias analysis to some extent during type propagation, the analysis is conservative and cannot trace through containers logic. For example, without function names, it is not possible to determine whether a call adds or perhaps removes elements from some C++ container.

One possible approach to narrow this gap is to learn and cluster patterns of generated assembly code for common containers (e.g., the standard C++ containers). Then, identify those patterns in the assembly of analyzed programs to map out the semantics of the containers and their functions. Identifying the functions is only the first step. In addition to that, the reference to the container must be traced through procedures in order to maintain the class types that the container stores. This becomes even more complicated with nested containers. Overall, precisely bridging such semantic gaps using only static analysis remains an open, very challenging, problem.

8.7.3 Virtual-dispatch-like C Calls

While I have not faced any false positives during my evaluation of VCI, it is possible that some non-virtual calls resemble the behavior of a C++ vcall dispatch. For example, VCI will incorrectly identify the following call as a vcall: a->b->foo(a), where a and b are pointers to plain C structs, and foo is a function pointer. It will also fail to find any constructor that defines the this pointer since the C struct types a and b will not have vtables. As a result, VCI will err in favor of security by limiting the target of foo(.) to any virtual function at the same offset of foo in b.

In vfGuard [133], the authors proposed a potential solution to this problem by looking for compiler-specific patterns in the assembly code. The authors argued that compilers tend to dispatch vcalls and nested C struct function pointers differently. However, based on my experimentation with the GCC compiler, there is no specific pattern that is used over the other. The authors of T-VIP [86] suggested recording the actual indirect call targets using a dynamic profiling pass that executes
benign test cases that (optimally) cover all indirect calls. Then, filter out misidentified vcalls if a recorded target is not in a vtable. However, the main challenge is in coming up with a conclusive benign input set that does not result in erroneous elimination and PFs at runtime. To the best of my knowledge, this remains an open research problem.

### 8.8 Destructors Corner Cases

The Itanium ABI defines three different types of destructors: 1. base destructor, which destroys the object itself, data members, and non-virtual base subobjects; 2. complete destructor, additionally destroys virtual base subobjects; and 3. deleting destructor, which in addition to performing a complete destruction, calls operator `delete` to free the object’s memory. Since base destructors do not call non-virtual bases, they do not reference any vtable and therefore are always ignored by VCI. Deleting destructors are also ignored since they call complete destructors and do not reference vtables. Complete destructors, on the other hand, have to call the virtual destructors of base classes. Therefore, they access the vtable of the object and its subobjects, in a somewhat similar behavior to constructors. Algorithm 8.2 implicitly assumes that all complete destructors are virtual. While that is true most of the time, there are a few exceptions to this rule.

For instance, the C++11 ABI added a `final` specifier that can be applied to classes. A class that is marked `final` cannot be inherited from (C++11 Clause 9.3). A final class can have a non-virtual complete destructor even though it defines or inherits virtual functions. This would cause VCI to incorrectly identify those destructors as constructors. However, the first thing a complete destructor does is store the vtable address of its class in the object’s memory. This is done before calling base destructors, if any. Therefore, VCI will not identify any base classes when analyzing the destructor site, compared to analyzing a constructor, when extracting inheritance relationships. VCI utilizes this disagreement in the identified “is-a” relationship to filter out non-virtual complete destructors (if any).

---

4The same approach could be utilized in augmenting the SameOff and AnyV policies by filtering out vcall targets that are never called by the benign inputs.
8.9 Cross-module Polymorphism

A C++ binary can use or inherit a class that is defined in a different module (shared library). In this case, space for the vtable of the shared class is reserved in the .bss section of the binary, but the contents of the vtable are not present until after the dynamic linker populates the .bss section. Similarly, in the case of cross-module inheritance, the derived class vtable may contain pointers to the PLT (Procedure Linkage Table), where the actual addresses of the base functions are to be determined at runtime by the dynamic linker. In both cases, VCI applies the SameOff policy since it has limited visibility into the shared vtables and the virtual function bodies. This gap can be narrowed via cross-module inter-procedural analysis, and a runtime stage, similar to VTV, that adjusts the policy as modules are loaded and the contents of the vtables become available. I leave this extension for future work.

8.10 Comparison to Related Policies

8.10.1 Reference Counts

It is possible to further strengthen the policies enforced by VCI via means of reference counting [168]. For instance, a vcall can never be invoked on a class type that has zero referenced instances. While this may result in additional reduction in the attack surface, the reference counters are vulnerable to memory corruption attacks since they have to reside in writable memory. Thus, VCI does not use reference counters.

8.10.2 Calling Convention

Though VCI handles the stdcall convention by default, developers could set specific calling conventions, such as thiscall andfastcall, for some virtual functions. This results in discrepancies in how arguments are passed to vcalls: stdcall passes arguments on the stack, thiscall passes only the this pointer in ecx, whilefastcall passes the first two arguments in ecx and edx. By identifying the calling convention at each vcall site, it is possible to filter out target virtual functions that do not adhere to the same calling convention. Care must be taken to precisely distinguish overlapping
conventions, such as thiscall andfastcall. This policy was applied by Prakash et al. [133], but it yielded minimal precision improvements (<1%).

8.10.3 Call Arity

In C++, polymorphs of a function must have the same parameters type list (C++14 Clause 10.3.2). This implies that they must also have the same arity, i.e., accept the same number of arguments. Therefore, it seems plausible to use the number of arguments passed to a vcall site to filter out potential target virtual functions that cannot accept that number of arguments. However, exact argument matching will be unsound, since at the binary level, only consumed parameters rather than accepted arguments are present. Additionally, as per the ABI, the this pointer is passed to class member functions regardless of whether the functions consume the this pointer or not. This discrepancy in the number of passed (prepared) arguments and the number of consumed parameters makes such policies unsound, as legitimate targets may be incorrectly eliminated if function polymorphs consume (use) a different number of arguments.

As a result, exact matching has to be relaxed by allowing compatible arguments, i.e., icall sites that prepare $N$ arguments can target functions that consume less than or equal to $N$ arguments. This policy was recently applied by TypeArmor, by van der Veen et al. [165], to protect indirect calls in both C and C++ binaries. TypeArmor, however, does not take the C++ semantics into consideration. Though the compatible arguments policy is sound, it is less precise than semantic-aware policies, as noted by the authors.

To evaluate how imprecise this policy is compared to VCI, I parsed the assembly dump of libxul.so, and counted the number of prepared arguments at each icall site as well as the number of accepted arguments by each function. Then, I computed the number of compatible target functions per icall site, grouped by the number of prepared arguments. Figure 8.7 depicts the results. For the sake of this argument, assume the best case scenario where any vcall site prepares only one argument (the this pointer). That means there are 188k compatible targets per vcall (functions that accept one or zero arguments). This is approximately 188× more targets per vcall than VCI. Even if, hypothetically speaking, policy refinements applied by TypeArmor would reduce that by 90%, there would still be

5The reported counts are underestimates as I ignored unused and variable length arguments.
18k targets per vcall, about $18 \times$ less precise than VCI. Hence, I conclude that generic policies based on call arity cannot replace C++ semantic-aware policies. This, of course, does not nullify the fact that layering multiple policies helps reduce the attack surface and is essential for complete protection at the binary level.

### 8.11 Summary

This chapter presented VCI, a system to generate and enforce a strict CFI policy against vtable attacks in COTS C++ binaries. VCI statically reconstructs various C++ semantics from the binaries without needing debug symbols or type information, making it applicable to any C++ application. VCI defeats vtable injection attacks and significantly reduces the attack surface for vtable reuse attacks. Experimental results on SPEC CPU2006 and Firefox showed that VCI is significantly more precise than state-of-the-art binary solutions. Testing against the ground truth from the source-based defense GCC VTV, VCI achieved greater than 60% precision in most cases, accounting for at least 48% to 99% additional reduction in the attack surface compared to the state-of-the-art binary defenses. VCI incurs a 7.79% average runtime overhead which is comparable to the state-of-the-art. I also discussed how VCI defends against real-world attacks and its impact on advanced vtable reuse attacks.
Chapter 9
Concluding Remarks and Future Directions

This dissertation has demonstrated the potential of automatic runtime application hardening by applying automatic techniques to successfully defend against DoS and CRA, the two largest vulnerability and attack classes. In addition, it has explored techniques to directly retrofit reactive security countermeasures into programs or into their runtime environment, giving direct control over the execution, detection, and prevention of attacks at runtime.

I have discussed Radmin [76, 78], a system for early detection of DoS attacks at the application level. Radmin monitored programs execution using Linux kernel tracing, and constructed PFAs that captured the patterns in the programs’ resource consumption sequences. The programs were then confined to their PFAs. I also presented Cogo [80] as a system to extend Radmin by providing linear training time and network I/O monitoring. Cogo enabled Radmin to scale to large server programs and offered practical features such as attaching to running processes and monitoring containerized programs.

To thwart CRA attacks, I presented EigenROP [77] as a transparent ROP detection system that focuses on statistical deviations in runtime program characteristics such as memory locality and reuse distance. Transparent detection is desirable in practice, since the defense does not interfere with the normal operation of the protected programs. While EigenROP incurred low overhead, the characteristics were monitored using heavy Pin based instrumentation, which ultimately resulted in a relatively high overhead overall. Finally, I presented VCI [79] as a CFI system for virtual call integrity in C++ programs. VCI reconstructed call semantics from binaries, and injected policies that prevented vtable corruption and injection, while significantly reducing the attack surface for vtable reuse. VCI had high accuracy and low runtime overhead.

One potential for future research is the construction of static input filters (firewall rules) to block inputs that may cause DoS. This may be achieved by encoding input invariants in the PFAs produced
from Radmin or Cogo. Exhaustion probability could be computed by traversing the PFAs on incoming inputs, without running the actual target program. Here, the PFA transition function would include input invariants, i.e., a transition edge would be taken if the incoming input satisfies the transition invariants. A byproduct of this would be the ability to estimate the resources needed to process the incoming input. This could have applications in cloud environments where resources are shared or allocated on demand.

It also should be possible to reduce the overhead incurred by EigenROP by implementing hardware run-time monitors agnostically, i.e., using universal execution models without including dependencies on internal hardware configurations. Hardware support also would increase detection accuracy by enabling low-cost, fine granularity monitoring. Both VCI and EigenROP depend on instruction set architecture (ISA) aware analysis. It may be possible to alleviate this by porting the analysis to an intermediate representation (IR) of the programs, such as VEX [127] or MIR [16].

In the experimentation discussed here, VCI was incomplete due to missed vcalls and overestimated vtables. Potential areas of improvement may include identifying the semantics of C++ containers, extending the analysis to support cross-module polymorphism, and conditionally executing pieces of code (e.g., using symbolic or forced execution [56, 104]) to identify and separate C-style invocations that might mimic vcalls. The analysis could also benefit from a dynamic profiling pass that augments the statically extracted semantics with semantics extracted from runtime traces collected using benign inputs.

So far, I have presented solutions that separately addressed DoS and CRA. Radmin and Cogo can accurately determine the resource consumption trace of a program, but they lack control flow information. On the other hand, EigenROP and VCI decipher control flow changes, but they are oblivious to resource flow. It may be possible to develop a hybrid model that combines information about both the control flow and resource consumption. The goal would be to enable accurate control and resource flow reasoning during program execution while achieving less overhead than layering separate defenses. It may be possible to approach this by studying the relationship between a CFG model [40] and an empirical resource control graph (RCG) [125] or the resource PFAs produced by Radmin. By combining these models into some form of a hybrid graph, the resulting graph would have edges that correspond to both control flow and resource consumption changes, and vertices that
correspond to control flow sources/sinks and resource usage information. Timing information could be added in vertices, and emitted empirical probabilities could be computed over transitions. I do recognize that this construction is involved and nontrivial. However, I believe that any realization of these hybrid models, even if applied selectively at only sensitive code sites in a program, could lead to very appreciable security gains.
Bibliography


Biography

Mohamed Elsabagh earned the doctorate degree in Computer Science at GMU in 2017. His research focuses on software and systems security, broadly spanning the areas of binary analysis, reverse engineering, mobile security, communications security, and machine learning. Before joining GMU, he earned a master's degree in Communication and Information Technology at Nile University, Egypt, in 2011, and a bachelor's degree in Computer Engineering at Alexandria University, Egypt, in 2009.