Integrity Postures for Software Self–Defense

Michael E. Locasto

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences
under the supervision of Dr. Angelos D. Keromytis

COLUMBIA UNIVERSITY

2008

ABSTRACT

Integrity Postures for Software Self–Defense

Michael E. Locasto

Software currently lacks the capability to respond intelligently and automatically to attacks in a way that preserves both its availability and its integrity. This problem is exacerbated by the occurrence of attacks that exploit previously unknown vulnerabilities or are delivered by previously unseen inputs. Systems that actively diagnose and repair themselves to become impervious, even when faced with such failures and attacks, are the holy grail of reliability and security research.

Unfortunately, most current software protection techniques typically abort a process after an intrusion attempt or a fault violates the integrity of an application's data or execution artifacts, thereby transforming an arbitrary exploit or failure into a self–induced denial of service attack — resulting in a compromise of both the application's integrity and its availability. Although many consider this approach safe, it is unappealing because systems remain susceptible to the original fault upon restart and risk losing accumulated state.

We provide novel mechanisms that preserve both the internal and external integrity postures of a software application so that it can continue to provide service in the face of a new attack or similar future instances of that attack. An integrity posture represents the disposition of a software system within a certain behavioral boundary. Preserving external integrity can prevent the recurrence of a particular fault or vulnerability, and it can be accomplished by filtering input that triggers or exercises the underlying error. Preserving internal integrity can be accomplished by making informed, targeted changes to a program's state in case a new attack is able to bypass the external integrity mediator.

In order to support the maintenance of both internal and external integrity postures, we introduce the novel notion of *policy–constrained speculative execution*, where the policy in question is an *integrity repair policy* based on our extensions to the Clark-Wilson Integrity

Model. Speculatively executing slices of an application, or *microspeculation*, is similar to hardware speculative execution, except that acceptance of a particular slice of execution is predicated on the integrity repair policy rather than the result of a branch conditional. These policies contain a collection of constraints that our system leverages to restore an internal integrity posture while an attack or fault occurs. Repair policy effectively customizes a software system's response to attack, providing an effective and novel means for software recovery.

We are the first to show that such an approach is feasible and practical by creating a runtime environment, STEM (Selective Transactional EMulation), that supervises an application's execution and enforces integrity repair policies. Our system operates in both binary–only and source–available environments, a major advance over previous work. Our performance evaluation using an experimental testbed that encompasses the use of both synthetic faults and real bugs shows that the system imposes from 1X to 2X performance slowdown during mainline operation and often completes repairs in a few tens of milliseconds. In order to preserve the external integrity posture of a software system, we combined STEM with an existing network content anomaly detector to create FLIPS (Feedback Learning Intrusion Prevention System), a system capable of automatic exploit signature generation. FLIPS generates exploit signatures in under a second and, as an unoptimized Java application, imposes a 23% overhead on traffic processing.

While no system provides perfect security, we can provide intelligent, well–formed recovery mechanisms and automatically invoke them. An integrity repair model assists in bridging the gap between current systems and systems that can automatically self–heal.

Contents

1 Introduction **1**

1.1 Current Threat Environment 1

1.2 Automated Response 3

1.3 Key Technical Problems 4

1.4 Hypothesis 6

1.5 Contributions 8

1.5.1 Integrity Postures 10

1.5.2 ROAR Workflow 10

1.5.3 Policy–Constrained Speculative Execution 10

1.6 Thesis Scope 11

1.6.1 Design Space 12

1.6.2 Implications 14

1.7 Organization 15

2 Related Work **18**

2.1 Traditional Defense Mechanisms 19

2.1.1 Protecting Control Flow 20

2.1.2 Artificial Diversity 21

2.1.3 Execution Supervision Environments 22

2.2 Automated Defense 23

2.2.1 Self–Healing 23

2.2.2 Survivable Systems 26

2.3 Anomaly Detection . . . 27
2.4 Exploit Signature Generation . . . 28
2.5 Vulnerability Signatures . . . 29
2.5.1 Capturing Vulnerabilities . . . 29
2.5.2 Systems . . . 30
2.6 Other Work . . . 33

3 Foundations of Integrity Postures **35**
3.1 The ROAR Workflow . . . 36
3.1.1 Stage 1: Recognition and Detection . . . 37
3.1.2 Stage 2: Orientation and Diagnosis . . . 37
3.1.3 Stage 3: Adaptation and Repair . . . 37
3.1.4 Stage 4: Response and Deployment . . . 38
3.2 Microspeculation . . . 39
3.2.1 Speculative Execution . . . 40
3.2.2 Policy–Constrained Speculative Execution . . . 40
3.3 The Clark-Wilson Integrity Model . . . 42
3.3.1 CW Model Background . . . 43
3.3.2 Model Limitations . . . 45
3.4 Constraint Satisfaction . . . 46
3.4.1 Theory of Constraint Satisfaction . . . 47
3.4.1.1 Definitions . . . 48
3.4.1.2 The Nature of Variables and Constraints . . . 48
3.4.2 RealPaver . . . 50
3.5 Error Virtualization . . . 52
3.5.1 Revisiting the EV Hypothesis . . . 53
3.5.2 Background: Memoization . . . 54
3.5.3 Evaluating the Range of Error Virtualization . . . 55
3.5.4 Experimental Setup . . . 57

3.5.5 Individual EV . . . 58
3.5.6 Summary . . . 61

4 Microspeculation Runtime Environments 64
4.1 Slice Selection Strategies . . . 65
4.2 Microspeculation Using Emulation: STEMv1 . . . 67
4.2.1 Instruction Set Randomization . . . 67
4.2.2 Selective Instruction Set Randomization . . . 69
4.2.3 The `objrand` Tool . . . 71
4.2.4 Derandomization and Detection . . . 74
4.3 Microspeculation Using Binary Rewriting: STEMv2 . . . 76
4.3.1 Core Design . . . 77
4.3.2 Supervision Coverage Policy . . . 78
4.3.3 STEM Operation . . . 79
4.3.3.1 Memory Log . . . 80
4.3.3.2 STEM_Preamble() . . . 80
4.3.3.3 STEM_Epilogue() . . . 81
4.3.3.4 SuperviseInstruction() . . . 81
4.3.4 Additional Controls . . . 82
4.4 Summary . . . 83

5 Maintaining an Internal Integrity Posture 84
5.1 Motivation . . . 84
5.1.1 Exception Shortcomings . . . 86
5.1.2 Problem: Semantics–Oblivious Healing . . . 87
5.1.3 Seeds of a Solution . . . 88
5.2 An Integrity Repair Model . . . 90
5.3 Ripple: A Repair Policy Language . . . 93
5.3.1 Repair Policy Overview . . . 94

5.3.2 Model of Computation 95
5.3.3 Repair Policy Execution 97
5.4 Assertion Use 98
5.5 Summary 99
5.5.1 Choosing Repair Scope 101
5.5.2 Discussion 101
5.5.3 Future Work 103

6 Maintaining an External Integrity Posture **104**
6.1 Automatic Signature Generation 105
6.2 FLIPS 106
6.2.1 Hybrid Detection 107
6.2.2 Proxying: Classification and Filtering 108
6.2.3 Feedback Learning 110
6.2.4 Limitations 111
6.3 Filter Generation, Enforcement, and Efficacy 112
6.3.1 Experimental Setup 113
6.3.2 Proxy Performance 113
6.3.3 Filter Efficacy 115
6.4 Expanding Exploit Filter Coverage 116
6.5 Countering Polymorphism 118
6.6 Summary 120

7 Behavior Profiling **122**
7.1 Observing Program Behavior 122
7.2 Profile Structure 123
7.3 Evaluating Profile Generation 126
7.4 Return Value Characteristics 131
7.4.1 Return Value Frequency Models 131

7.4.2 Clustering with k-means . . . 133
7.5 Limitations . . . 133
7.6 Summary . . . 134

8 Evaluation **139**
8.1 Microspeculation . . . 139
8.1.1 Coverage vs. Performance . . . 139
8.1.1.1 Slice "Work" Measures . . . 141
8.1.1.2 Macrobenchmarks . . . 146
8.1.1.3 Coverage Summary . . . 150
8.1.2 Performance of STEMv1 . . . 153
8.1.3 Performance of STEMv2 . . . 154
8.1.3.1 Experimental Setup . . . 155
8.1.3.2 Microbenchmarks . . . 156
8.1.3.3 Performance Without Startup Penalty . . . 158
8.1.3.4 Summary . . . 160
8.2 Repair Policy . . . 161
8.2.1 Synthetic Vulnerabilities . . . 161
8.2.1.1 Stack Overwrite Example . . . 161
8.2.1.2 Apache Stack Overwrite Example . . . 162
8.2.1.3 Authentication Skip Example . . . 162
8.2.2 Wilander Testbed . . . 164
8.2.3 Analyzing Real Vulnerabilities . . . 173
8.2.3.1 libpng . . . 174
8.2.3.2 NULLhttpd . . . 176
8.2.3.3 fetchmail . . . 179
8.3 Future Work . . . 179
8.4 Summary . . . 180

9 Discussion **183**

9.1 Limitations . . . 183

9.1.1 Automatic Repair Policy Generation . . . 184

9.1.2 Speculated I/O . . . 184

9.1.3 Kernel–Level Supervision . . . 187

9.1.4 Automatic Repair Validation . . . 187

9.2 Research Opportunities . . . 189

9.2.1 Applicability of Repair Policy . . . 189

9.2.2 Pair Programming . . . 190

9.2.3 Application Communities . . . 191

9.3 Musings . . . 191

9.3.1 Ethical Considerations . . . 192

9.3.2 Attacker Intent . . . 193

10 Conclusion **194**

10.1 Thesis Summary . . . 195

10.2 Results Summary . . . 196

10.3 Closing . . . 199

A Ripple **200**

A.1 Ripple Language Tutorial . . . 200

A.2 Ripple One-Liners . . . 208

A.3 Planned Improvements . . . 209

Bibliography **210**

List of Figures

3.1 Speculative Execution of a Branch 41
3.2 Constraint with Arbitrary Control Flow 51
3.3 Example RealPaver Program 51
3.4 Error Virtualization Example 52
3.5 Post-EV Correctness for Individual `true` Functions 59
3.6 Post-EV Correctness for Individual `hostname` Functions 61

4.1 STEMv2 Architecture 65
4.2 STEM Version 1 (emulator) API 68
4.3 Randomization of the Instructions of a Program Slice 70
4.4 An Example `objrand` Configuration File 72
4.5 Sample Execution of `objrand` 74

5.1 Handling a "simple" Exception 86
5.2 Semantically Incorrect Response 88
5.3 Integrity Repair Model Pseudocode 93
5.4 An Example Repair Policy 95

6.1 Architecture of FLIPS Prototype 107
6.2 Exploit Filtering 109
6.3 Exploit Signature Generation 110
6.4 STEM Providing Feedback to FLIPS 111

6.5 Performance Impact of FLIPS Proxy 114
6.6 Dynamically Generated Example Exploit Filter 121

7.1 Computing Execution Window Context for a Profile 124
7.2 Drop in Average Valid Window Context 125
7.3 Computing Return Value Predictability Score 126
7.4 Average Predictability of Return Values 127
7.5 Return Value Predictability for both `wget` and `httpd` with Different Window Sizes 128
7.6 Per-Function Return Value Predictability for both `wget` and `httpd` 129
7.7 Relatively Scaled k-means Clusters for `sort` 135
7.8 Return Value Frequency Distributions for a Compression Program 136
7.9 Return Value Frequency Distributions for Output Programs 137
7.10 Return Value Frequency Distributions for Hash Programs 138

8.1 Work Characterization (Instructions per Slice) 142
8.2 Work Characterization for `/bin/true` in Terms of Cumulative Write Size . 143
8.3 Relationship Between Instructions per Slice and Total Memory Write Size per Slice for `/bin/true` 144
8.4 Work Characterization for `/bin/arch` 145
8.5 Work Distribution for Various Utilities 147
8.6 Work Distribution for two `/bin/sh` Sessions 148
8.7 Work Distribution for `gzip` 149
8.8 Work Distribution for `md5sum` 151
8.9 Work Distribution for Apache 152
8.10 STEM Version 1 (emulator) Performance Range 154
8.11 Example Repair Policy 162
8.12 Repair Policy for Apache Sample Vulnerability 163
8.13 Invoking `authskip` with Incorrect Credentials 164

8.14 Invoking `authskip` with Correct Credentials 164
8.15 Invoking `authskip` under STEM 165
8.16 STEM Repairing `authskip` 166
8.17 Sample Wilander Repair Policy 172
8.18 KView Using Fixed libpng 175
8.19 Exploiting libpng 176
8.20 Repair Policy for libpng 177
8.21 Repair Policy for nullhttpd 177
8.22 Repair Policy for fetchmail 179

List of Tables

3.1 Correctness Scale for Error Virtualization 57
3.2 Error Virtualization Microbenchmarks 58
3.3 Distribution of Correctness Results for `true` 60
3.4 Distribution of Correctness Results for `hostname` 60
3.5 Distribution of Correctness Results for `arch` 62
3.6 Distribution of Correctness Results for `sort` 62
3.7 Distribution of Correctness Results for `echo` 63
3.8 Distribution of Correctness Results for `md5sum` 63

5.1 Assertion Statistics 99
5.2 Key for Assertion Table 100

6.1 Performance Impact of FLIPS Proxy 115

7.1 Percentage of Unpredictable (Outlier) Functions 130
7.2 Manhattan Distance Within and Between Models 132

8.1 STEM Slice Profiling Microbenchmarks 141
8.2 STEM Version 1 (emulator) Microbenchmarks 155
8.3 STEM Version 2 (binary rewriting) Microbenchmarks 157
8.4 STEM Version 2 (binary rewriting) Impact on Apache 158
8.5 Wilander Testbed Index 167
8.6 Wilander Testbed Control Sample Results 169

8.7 Wilander and STEM With and Without Repair Policy 171
8.8 Timing Information for Wilander Repairs 173

This thing of darkness I / Acknowledge mine. – Prospero, 5.1.278-279, The Tempest

Acknowledgements

Why the quote from Prospero? In reading the play, we understand that the quote is one that expresses *responsibility* rather than *ownership*.

The process of producing a thesis has a deep and lasting influence on those who labor to create one. As the philosopher shapes the thesis, so the thesis shapes the philosopher. This "thing of darkness" is not necessarily negative. It is an influence that is strange and unknown: an influence that gradually becomes a familiar part of you. The result is a fundamentally different view of the world, but a view that is a synthesis of what you have been and what you have learned.

Thanks are often given casually for the meanest of things, and rarely at the point of most gratitude. I can only hope that a mention here makes up for not saying it often and sincerely enough all the folks listed here (and some that are not) that deserve it and much more from me.

The LORD gave me the talent to use, abuse, and misuse, and carried me through things that I could not handle. My lovely, gentle, sweet-natured wife, **Kim**, whose love and emotional support started well before this entire process, is the only one entitled to what comes out as a result. **Angelos Stavrou** shared the difficulties and triumphs of the process. Angelos is the rare person who is direct and honest above all else. **Eleazar Eskin** told me, mere moments after I arrived in NYC and mere moments before he departed, that the PhD was mostly baloney and about 15 minutes of real, actual hard work. I leave it as an exercise to the reader to find out if he was right. **Matthew C. Little** is a role model and my closest friend. I cannot say enough about him, so I'm not going to try. If you know him, your life is better. Matthew also read every word of the thesis and offered constructive comments and an amusing perspective on various portions of the document.

My siblings, **Regina**, **Christopher**, **Steven**, and **Charles** will always understand each other. Thanks especially to Christopher, for dodging bullets in Iraq while I blissfully enjoyed our rights and freedoms here at home. **Michael J. Hulme** is a good friend whose temperament nearly mirrors and talent clearly exceeds my own. I particularly enjoyed how

he motivated me to finish Stage 4 (arbitrary control flow for an 8088 assembler) in Rebecca Mercuri's (the Rebecca Mercuri who wrote the seminal dissertation on electronic voting) computer organization course at TCNJ. **Ryan Gladisiewicz** and **Justin Tracy** deserve a special mention as two good friends of mine who share a unique and whimsical perspective on life and how to refrain from being uptight. I really enjoyed Ryan's company on a cross country drive from Utah to El Paso to New Jersey in the Spring of 2005. **Matt Hall** has been a very good friend to both Kim and me.

My adviser, **Angelos D. Keromytis**, is a constant source of creativity, inspiration, and encouragement. Angelos has taught me most everything I know about managing an academic career; he's adept at it and makes it look easy, even though it takes years of intense, sustained effort. Angelos has offered me unflagging support, freedom, and friendship over the past five and a half years. He deserves the lion's share of credit for enabling this thesis to even exist, and his meticulous editing and comments on a draft greatly improved the final product.

Quite a few people have contributed in various ways to strengthening the intellectual merit of this work. **Stephen W. Boyd** was instrumental in porting a large part of Bochs to provide the basis for STEMv1. **Gaurav Kc** provided me with his version of `objrand`, which randomized whole binaries. I used it as a guide to creating my selective `objrand` and merged the two. **Marcus Peinado** and **Helen Wang** worked closely with me during the summer of 2006 to create the techniques for generalizing exploit signatures. **Steve Bellovin** pointed out that repair policy has the benefit that it can be shut off without greatly impacting the normal operation of an application — unlike a patch. Angelos Stavrou and **Gabriela Cretu** helped produce the method of predicting return values and created the prediction graphs. Gabriela was instrumental in producing the assertion survey, and she also read a draft of this thesis and helped take care of some of the paperwork issues when I could not deliver it in person.

Sal Stolfo has been a mentor of unparalleled value to me. I cherish his advice and guidance. I especially thank Sal for reading this thesis and being on the committee during

his recuperation.

Anil Somayaji was really the first external person to ask me the direct question of what problem I was solving when we talked at USENIX Security 2006. This question, conversation, and my attempt to answer it really was the beginning of the end of the process of producing this dissertation. It got me into gear and focused my thinking.

Sal and Angelos also deserve credit for sitting me down and pushing me over the "writing edge" at the end of August 2007; previous to that episode, I had been procrastinating about any sort of continuous, dedicated writing effort. In addition, at that time, Angelos Stavrou shared his advice for actually getting the writing done. It worked.

I heartily thank my thesis committee, including **Dan Rubenstein**, Angelos, Sal, Steve, and Anil for reading and commenting on yet another yawning (and potentially yawn-worthy) tome of a thesis.

This thesis would not have been possible without the previous support, encouragement, and guidance I received during my early education. Coach **Jeff May** (who also served as my Honors Chemistry teacher), whom I have tremendous respect for, introduced me, at the not so tender age of 14, to the reality of hard work and discipline. **Mary-Lou Grady** was an early sponsor and aggressive advocate of my education, and welcomed me into the Odyssey of the Mind after-school competition during my years at St. Peter of Alcantara in Port Washington, Long Island. When my self–esteem shrank and my self–doubt grew, she was there to cheer me on and steer me back on course. **Robert Johnson** is a superbly gifted and talented teacher; he taught me computer science, mathematics, and programming in my latter two years of high school and managed to impart some sense of maturity and responsibility during a volatile period. Taking Honors English 3 with **Ralph Caiazzo** — and the decision process to take the entrance test for it — literally changed my life as well as contributed greatly to helping my writing style mature. Yes, there is a story here.

Ursula Wolz, herself a doctoral graduate of Columbia's CS Department, was my undergraduate adviser and mentor at TCNJ. I appreciate all her advice and encouragement over the years and the freedom she allowed me to develop my research skills as an un-

dergraduate. **Pete DePasquale** (currently a tenured assistant professor at TCNJ) and **Mike Massimi** (currently a PhD student at the University of Toronto) have been valuable collaborators and friends.

The folks I have known in the NSL and IDS labs were a pleasure to work and play with, especially **Matthew Burnside**, Gabriela, **Wei-Jen Li**, **Janak Parekh**, **Stelios Sidiroglou**, **Yingbo Song**, and **Ke Wang**. Working with Angelos Stavrou and Gabriela on various projects (among them SEV, STAND, and Omnisense) was a fun and rewarding process, and I thank them for their friendship. I also enjoyed working with Gabriela, Matt, and Stelios on the Snakeyes project during the summer of 2005. I was delighted to get to know a lot of other folks in the NSL, IDS, and PSL groups, including **Elli Androulaki**, **Brian Bowen**, **Rean Griffith**, **Phil Gross**, and **Angelika Zavou**. In addition, the CRF staff, especially **Daisy Nguyen**, **John Petrella**, and **Darrell Bethea**, were a pleasure to get to know.

The city of Vancouver, British Columbia is a place of unparalleled natural beauty in which to undertake the pressures of a dissertation.

Michael E. Locasto
August 27, 2007 — May 14, 2008
Vancouver, BC, Canada
New York, NY, USA
Hackettstown, NJ, USA
Lebanon, NH, USA

Distributed 12 November 2007;
Defended 17 December 2007;
Minor Revisions Accepted 16 May 2008

Chapter 1

Introduction

This thesis describes the application of machine intelligence to the problem of runtime software self-defense. Software self–defense mechanisms seek to automatically remedy the effects of a fault or vulnerability so that execution continues safely while maintaining system availability. The novel mechanisms that we introduce are primarily concerned with helping software systems maintain availability through the use of *integrity postures* to ward off attacks — particularly those that exploit memory corruption vulnerabilities. Whereas traditional software protection techniques typically terminate the "protected" process in response to attack, our work protects systems by both blocking malicious input and recovering to a safe execution flow.

1.1 Current Threat Environment

Current attacks on software systems are automated, occur quickly[1], often remain unnoticed by the system owner, and exploit a wide variety of common vulnerabilities and programming errors. In addition to more traditional attacks on network–facing services like DNS, HTTP,

[1] Although complicated attacks may consist of multiple steps with arbitrary delays between each stage of the attack, at a basic level, each step of the attack proceeds at machine speed: the rate at which the hardware can execute instructions. This speed is, in any event, much faster than humans can make a decision.

FTP, and SSH, many attacks are embedded in or delivered via "rich" content, such as Javascript, ActiveX, Flash, or other lightweight web programming environments. Simply visiting a website[2] or viewing a file icon in a directory (the WMF attack is but one example of this risk) is enough to trigger an exploit and cause an infection or a compromise.

Attacks and exploits can be embedded in many types of content, and it is often not immediately apparent, even to a specialist, why a particular packet, message, image, or web object is malicious. Users cannot be expected to examine binary streams of image or movie content, or deconstruct obfuscated Javascript code to determine if the content in question poses a risk to their computing platform. Furthermore, the sheer number of services, applications, and data that an administrator or computer owner must protect quickly overwhelms any sort of manual supervision capability that may exist.

Users are continuously presented with the same careworn advice when faced with computer security problems. This advice consists of platitudes about ensuring that an anti-virus signature database is updated, that the latest application and operating system patches have been installed (potentially requiring a system or application reboot during the process), and that "care" is taken when visiting websites or opening emails from untrusted sources. Such advice places a heavy burden on the shoulders of largely untrained computer users with more pressing tasks to accomplish than figure out what a DLL is, where it belongs, and whether or not the popup message carrying a missive about `something.DLL` is causing their music player to skip, when, minutes ago, everything was fine — right before they installed that critical security update. Users may rarely perceive any real damage due to a breach of security. They are typically more sensitive to the side effects stemming from an intrusive piece of security software meant to guarantee their safety, security, and peace of mind.

It is unlikely that system security problems will ever completely disappear. New and creative exploits continue to emerge and take advantage of mistakes in system design, construction, configuration, and deployment. It is difficult, if not impossible, to perceive or predict all threats *a priori*. Furthermore, many exploits are launched with little or no warning, and the window to protect systems against *known* vulnerabilities (only 5.8 days in 2004) has been shrinking for some time [TE04]. Current estimates are much shorter,

[2]Such attacks bear the evocative moniker "drive-by downloads."

and the Zotob worm was released 3 days after the vulnerability was announced. The job of security professionals and system administrators is not getting any easier. In the case of worms like Slammer, malware spreads so quickly as to defy meaningful human intervention. In order to have a reasonable chance of surviving or deflecting attacks — especially common attacks that exploit memory corruption vulnerabilities — a system must incorporate automated self–defense mechanisms.

Therefore, a key challenge in computer security is for systems to automatically react to and protect themselves from attacks. This fundamental problem — the inability to automatically mount a reliable, targeted, and adaptive response [Ove98] — is magnified when exploits are delivered via previously unseen inputs[3] or exploit previously unknown vulnerabilities. In the first case, no signature matching the attack is available. In the second, no patch is available from the software vendor.

1.2 Automated Response

Since many attacks are automated, and users cannot hope to deal with the wide variety of existing and future attacks manually, it appears that defense systems must also be automated. Automation of defense mechanisms offers quick response times and scalability (in terms of handling many attack instances at once) and thereby offloads or reduces the burden on system owners and administrators. A speedy but incorrect response, however, is no substitute for an accurate one. Despite the perceived benefits of automated response, the unpredictable nature of attacks, the imprecision of existing detection mechanisms, and the dearth of analysis and guarantees for automatic response systems has led to a lack of confidence and a justifiable degree of skepticism about automated defense techniques.

Automating a response strategy is difficult, as it is often unclear what a program or system should do in response to an error or attack. Almost by definition, exploited vulnerabilities represent unanticipated error conditions in a system. A response system is forced to anticipate the intent of the programmer, even if that intent was not well expressed, well–formed, or absent altogether. Ideal computing systems would recover from attacks and

[3]Such attacks are sometimes referred to as “zero day exploits.”

errors without human intervention. However, the state of the art is far from mature, and most existing protection mechanisms forcibly terminate the process that was attacked (and do nothing to fix the fault, thereby ensuring that the system is still vulnerable when it is rebooted or restarted).

The limitations of detection technology have historically mandated that the shortcomings of intrusion detection (false positives and negatives, fail–open nature, performance, *etc.*) be addressed before reaction mechanisms are considered — an attack must be detected before any response can be mounted. In addition, many system administrators are understandably reluctant to allow an automated defense system the latitude to enact unsupervised changes to the computing environment, even though (and precisely because) a machine can react orders of magnitude faster than a human.

This risk, however, does not change the fact that the current generation of widespread software systems defense tools do little to protect the majority of users against unanticipated faults and vulnerabilities in the wide array of complicated, massive (and often premature and incomplete) software they use on a daily basis. Even the software users normally do not think about (*e.g.,* the infrastructure supporting online banking, the purchase of our movie or plane tickets, and news services) is rife with undiscovered problems. The fact remains that damage *is* done: reported business losses from large–scale network attacks sometimes figure in the billions or hundreds of millions of dollars for the industry slice or economy as a whole. For end users, files disappear, once–reliable machines exhibit unexpected behavior patterns, optical media drives inexplicably open and shut, popup messages litter the desktop, disk space disappears, and (invariably) the Internet "slows down."

1.3 Key Technical Problems

There are a number of technical obstacles to overcome on the way to creating a software self–defense system. The underlying research question is: when an application receives and processes input from an attacker that compromises the application's independent operation and integrity, what exactly should the application do, given that continued availability is a desirable property?

Even after observing a series of attacks, current systems can neither survive the attack without remaining under the control of the attacker, nor can they automatically filter out the type of input responsible for allowing the attacker access in the first place. In effect, they generate neither a remedy nor a prophylactic measure.

State-of-the-art research on intrusion defense mechanisms typically responds to an attack by terminating the attacked process. Even though process termination is considered "safe", we see this approach as unappealing because it leaves systems susceptible to the original fault upon restart and risks losing accumulated state. Furthermore, it remains unclear whether this approach is, in actual fact, safe: for example, crashing in the middle of a database update may be quite risky.

Instead, many research efforts seek to stop malicious input before it reaches an application. Automatically creating reliable signatures of zero-day exploits is the focus of intense research efforts. Signatures of viruses and other malware are currently produced by manual inspection of the malware source code. Involving humans in the response loop dramatically lengthens response time and does nothing to stop the initial attack. In addition, because even standard defensive instrumentation usually imposes non–negligible performance costs, it is often desirable to automatically generate either vulnerability [CCZ+07, CCCR05, NBS06, CPWL07] or exploit signatures [LWKS05, SC05, LS05, XNK+05] to prevent malicious input from reaching and subverting the protected system or unnecessarily exercising the self–healing instrumentation. Novel contributions of this thesis include both a system that accomplishes the latter type of automatic signature generation and filtering [LWKS05] and a technique for supporting the generation of generalized exploit signatures that is similar to the one reported by Cui *et al.* [CPWL07].

Problem Summary In short, unanticipated error conditions such as those stemming from faults or attacks cause downtime for computers. Attackers are highly motivated to seek out and exploit software weaknesses for financial gain, personal renown, or revenge. Attacks occur at machine speed. Defenses must be automated, but weaknesses and vulnerabilities cannot be precisely predicted or completely eliminated before deployment. Therefore, systems need automated, general mechanisms for runtime repair, but current techniques for

software self–defense do not include correct remediation mechanisms.

1.4 Hypothesis

It is our belief that software systems can defend themselves. We aim to help software defend itself against attacks that are encoded in previously unseen inputs and that exploit previously unknown vulnerabilities without relying on crashing as the state of the art in response techniques. The benefits of such a capability include decreased worry and effort on the part of the human users of the system and increased availability of the system's services. We will demonstrate that it is practical to create and apply mechanisms for automatically strengthening both the internal operation and the external input boundary of a software system so that it cannot be made to fail in the same way again. To do so, we need to invent a mechanism that can supervise the software and automatically enact a correct remedy to ensure continued availability as well as automatically generate generalized exploit signatures to prevent the reintroduction of malicious input to the system. Our notion of software self–defense relies on our invention of a novel supervision mechanism: selective, policy–constrained speculative execution, or *microspeculation*.

Because self–defending software necessarily assumes the absence of a human or human decision from the critical path of repair, we need to supply a mechanism for automated reasoning that can select or generate an appropriate response. We employ an existing form of machine intelligence, constraint satisfaction, to fix a software system "on the fly." We invent the notion of *repair policy* to express those constraints in a language we call Ripple[4].

Thesis Statement: *This dissertation examines the claim that it is practical to speculatively execute a software system subject to an integrity repair policy so that faults and exercised vulnerabilities are effectively and automatically remedied by machine intelligence to preserve both the internal and external integrity postures of a software system. In addition, microspeculation–based supervision can help generate exploit signatures to protect the system input boundary.*

[4] "Ripple" sounds like RPL, the acronym for Repair Policy Language.

Consider a program P and a set of inputs, D (*e.g.,* a series of network packets, protocol messages, file data, or user input). Some subset of that input data contains a set of exploit data e, where, for example, individual members of e may be the bytes of a malcode instruction sequence. We focus specifically on code injection attacks that exploit memory corruption vulnerabilities.

$$\{e_0, e_1, \ldots, e_n\} \subseteq D \tag{1.1}$$

When P is run and receives input D containing exploit data, it enters an exploited or error state σ (according to some fault detection mechanism or intrusion sensor).

$$\text{Run}(P, D) \rightarrow \sigma \tag{1.2}$$

After P consumes D and reaches the exploited or error state σ, a self–healing function H operates on P and σ to produce a "healed" or repaired program P'. We hypothesize that such repair can be achieved (*i.e.,* both an internal and external integrity posture re-established) by supplying a repair policy R to H and using H to generate an exploit signature F to filter out future instances of the attack. A repaired internal integrity posture involves using R to both restore the integrity of critical data items and (optionally) replace σ or other states with correct or healed versions. A repaired external integrity posture involves using F to filter or reject input data similar to $\{e_0, e_1, \ldots, e_n\}$ to preserve the input boundary.

$$H(P, \sigma, R) \rightarrow P', F \tag{1.3}$$

The repaired program P' includes both the input filtering and the activation of a mechanism for self–healing. The purpose of an exploit signature generation algorithm is to identify the subset $e_0, e_1, \ldots, e_n$ of D and ensure that:

$$\text{Run}(P', \{e_0, e_1, \ldots, e_n\}) \not\rightarrow \sigma \tag{1.4}$$

that is, to prevent the healed version of the application from entering the error or exploit state on replay or reintroduction of the attack data.

We will show that it is possible to create software that uses machine intelligence (specifically, constraint satisfaction) to defend itself from attacks and other unanticipated faults. Specifying the solution state for constraint satisfaction remains the hard problem, and we leave that problem squarely in human hands. We intend to investigate methods of automated repair policy creation in follow-on work.

Introducing repair policy helps solve the problem, not just move it around. We split the problem of automatic repair into a policy specification problem (something humans can be good at) and a policy satisfaction problem (something computers are good at). As a result, we automate a large, mechanical part of the problem that needs to be solved during runtime, when the system is on the critical path of self-healing. We leave as a manual exercise the task of policy specification — a task that can benefit from human expertise and is not on the critical path of runtime, real-time repair.

1.5 Contributions

During our examination of the central hypothesis, this thesis makes several novel contributions to the field of information systems security in general and software self–healing in particular:

- We are the *first* to propose and implement the concept of *microspeculation*: selective, policy–constrained speculative execution in which execution can seamlessly alternate between supervised and unsupervised execution. When we began our research, no platform existed that allowed such flexibility.

- We provide the *first* comprehensive software self-defense system based on a combination of external and internal integrity postures: a system that simultaneously attempts both self–healing and filter generation & enforcement.

- We describe a new and practical method of automatically (*i.e.,* completely without human supervision) generating exploit signatures to filter out *previously unknown*

attacks to help maintain an external integrity posture.

- We provide extensions to the Clark-Wilson Integrity Model. These extensions provide the basis for the novel concept of *repair policy*, a new mechanism for maintaining an internal integrity posture and controlling microspeculation.

Microspeculation provides a defense system with a clearer view of the impending disposition of a monitored system by the use of speculative execution. An internal integrity posture governs the integrity of critical data items within a system. An external integrity posture governs the integrity of a system's input boundary. To use a biological analogy, this dual approach to self–defense simultaneously thickens the skin and boosts the immune system.

We also introduce a number of specific novel supporting mechanisms that help increase the power or generality of these high–level research contributions.

- We define a workflow (ROAR) that helps organize the process of automated repair and software self–defense.

- We build a prototype of a noninvasive platform and supporting runtime environment ideal for microspeculating real, large, commonly used software applications and conducting research into methods of evaluating automated system defense and self–healing.

- We augment our microspeculation system with a mechanism for estimating the workload distribution of microspeculated programs.

- We augment our microspeculation system with a test harness for examining the behavior of a program after it has been subjected to a self–healing strategy like error virtualization [SLBK05].

- We present a method of generalizing our exploit signatures by mapping them to parts of a file format or network protocol message.

- We provide a practical form of Instruction Set Randomization [KKP03].

1.5.1 Integrity Postures

An integrity posture represents the behavioral integrity of a software system. We introduce two novel mechanisms for maintaining the internal and external integrity posture of a system, respectively. The first, *repair policy*, contains a collection of constraints that a system can use to repair or self–heal after it detects a fault or attack. The second mechanism is a novel exploit signature generation and enforcement system. An integrity posture represents the disposition of a software system within a certain behavioral boundary.

1.5.2 ROAR Workflow

The ROAR workflow consists of four stages that we consider necessary to enact a self–healing repair. Systems that employ ROAR first (*a*) *Recognize* a threat is present or an attack has occurred, then (*b*) *Orient* the system to this threat by analyzing it, next (*c*) *Adapt* to the threat by constructing appropriate fixes or changes in state, and finally (*d*) *Respond* to the threat by verifying and deploying those adaptations. We give more detail on ROAR and how it relates to the OODA (Observe, Orient, Decide, Act) loop in Chapter 3.

1.5.3 Policy–Constrained Speculative Execution

Microspeculation adds a policy-driven layer of indirection to the execution of a software system's instruction stream in order to intercept and examine the actions of the process before they become "committed" or otherwise visible to external parties. Program interposition or instruction interception provides the basis for techniques like sandboxing, system call interposition [sub02, FBF99, SF00, Pro03], and `chroot`-style jails.

These approaches differ from microspeculation because they do not speculatively execute the supervised process. In addition, they usually occur at static, well-defined program execution points, such as system calls. Most importantly, they only seek to detect or contain the damage, not prevent it from occurring — nor do they provide any way to remedy the underlying fault or vulnerability.

Our microspeculation runtime environment can be used as a platform for conducting research into a variety of software self–healing and collaborative security mechanisms.

1.6 Thesis Scope

While the *problem* of software self-defense is not new, it remains a very important one. The integrity of software that supports critical infrastructure like medical, government, financial, and military systems has far–reaching implications for society at large. The solution we present in this thesis *is* new, and it supports the capability to self–heal at runtime without direct (and slow) human involvement. The novel aspects of this thesis are found in the contributions we list above. The area of software self–healing can be considered closely related to the autonomic computing initiative (also known as self-* systems) espoused by IBM[5]. That effort, in contrast to our own, is primarily aimed at reducing the management burden and downtime of large networked systems that provide computation as a service. Autonomic computing largely remains a goal or vision rather than a fully deployed series of techniques. The techniques that this thesis presents can be used to help advance the field of autonomic computing.

We must also be careful to delineate the range and class of failure conditions our system is meant to handle. Clearly, the system is not immediately appropriate for social engineering attacks. No system can address all attacks, and some failure modes may be indistinguishable from normal operation. This thesis is not about novel detection techniques *per se*. We are primarily interested in what should happen to a process *after* the detection of an attack, fault, or exploit occurs.

To keep our work grounded (and to avoid claiming the "merely possible" as the "already accomplished"), we focus primarily on a pervasive, major threat to computing systems: binary code injection attacks[6]. The examples and prose throughout this thesis reflect this bias. We would note, however, that our integrity repair model is meant to remain agnostic to the particular type of failure or attack. If a sensor can be written to algorithmically detect the symptoms of an attack or fault (such as non-deterministic timing bugs, misconfiguration,

[5] `http://www.ibm.com/autonomic/`

[6] Certain detection and protection techniques, such as Instruction Set Randomization (ISR) and tainted dataflow analysis can prevent a wide variety of injection attacks, but they result in the process crashing — something we explicitly wish to avoid. STEM uses ISR as a sensor, and we consider the limits of ISR later in Chapter 4.

communications failure, and DoS) or if the symptoms of the attack manifest as violations of the integrity policy, then our system should repair the fault.

1.6.1 Design Space

Part of the novelty of the thesis is in the synthesis of lightweight binary supervision, automated repair, and exploit signature generation. The resulting system is comprehensive and transparent. These two properties help argue for the practicality of the proposed system.

Our system reflects a particular combination of parameters in the design space of host–based software self–defense mechanisms. There are several degrees of freedom to consider.

1. *Host-based vs. Network-based.* Is the system host-centric or does it operate on the network? If at the network, does it operate on the LAN, WAN, or Internet scale, and does it handle encrypted traffic?

2. *Transparency.* How invasive is the system in terms of its impact (performance, stability) on the application software?

3. *Support for Legacy Software.* Does the system provide support for and easily integrate with the large legacy base? Can legacy systems be executed unmodified?

4. *Performance.* Although performance may be affected by the system level placement, does the system in general add a performance penalty? If so, where does the penalty manifest, and how large is the impact? What is the trade-off between protection and performance?

5. *System Level.* Is the system placed at a particular level in the system hierarchy or split between two (or more) levels? Possible places to insert the system are raw hardware, the architectural level, the operating system kernel, a kernel service thread, a virtual machine monitor (VMM), a user-level library, the application software itself, third-party applications, or remote monitors.

6. *Implementation Difficulty.* Is the system difficult to implement or reverse engineer? Are technical aspects generalizable or are they narrowly applicable and difficult to

reproduce? Does the complexity of the system defeat attempts at verification and analysis?

7. *Interoperability.* Does the system provide an architecture that is modular? Can it easily interact with similar systems and components written by third parties? Does the system use proprietary communications protocols or does it adopt accepted standards?

Our system represents a vector in this space. In particular, the system:

- **is a host-based supervision mechanism.** STEM, our supervision environment and research platform [LSCK07, SLBK05], focuses on protecting a single application from attack or compromise, although the information it gathers can be distributed to other peers or interested parties as part of an Application Community [LSK06a, LSK05]. Our external integrity posture system, FLIPS [LWKS05], provides exploit signatures for network–facing applications.

- **is minimally invasive for the application.** Supervised software applications need not be recompiled. The operation of the monitoring and remediation is not visible to applications, but may be visible to the OS. The system *may* export a means for exercising control or observation of these mechanisms to the application via the repair policy.

- **contains support for the legacy software base.** Supervision of legacy applications and COTS software is supported.

- **is performance sensitive.** Instruction supervision and execution remains at native hardware speed after the initial instrumentation phase.

- **is placed between the supervised application and the operating system.** STEM intercepts user–level instructions before the OS is invoked. From the OS perspective, STEM is part of the supervised process. From the perspective of the guarded software application, STEM is a transparent service.

- **has a moderate-to-hard implementation difficulty.** In order to obtain a system satisfying the other requirements, implementation difficulty is relatively high and

requires substantial design work. Our burden is somewhat eased by the adoption of an existing constraint solver and an existing binary rewriting mechanism. Our system can benefit from independent performance and efficacy improvements in these systems.

- **provides a modular and flexible architecture.** The system can "plug-in" different sensors and supervision coverage strategies. The system's repair response is flexible because the repair is controlled via an externally specified policy.

1.6.2 Implications

We expect the work presented in this thesis to have an impact on several areas of software systems engineering. We believe that our work on microspeculation and repair policy can change the way software is:

- designed and developed — by adopting a program structure that is amenable to repair
- tested — by leveraging the repair policy language to perform controlled fault injection
- updated — via non–invasive repair policies
- executed — via the use of lightweight binary supervision

Microspeculation and repair policy can impact the way software systems are designed and developed by providing a model which developers and system designers can emulate or aspire to when constructing their systems. They can make the structure of their programs amenable to this style or model of repair. We expect that doing so for future systems would improve the efficacy of these approaches. Although complete adoption is probably not achievable, codifying the model into best practices would have tangible benefits. Repair and self–healing are simply other *aspects* of software development, and if developers can design their software in a standard way, we can provide repair as a service of the runtime environment.

Microspeculation is, at its heart, an argument for changing the way software is executed. Such an argument has been building for some time with the advent of large main memory

capacity, faster CPU clock cycles, deep superscalar pipelines, hyperthreading, and multi-core and many-core CPU architectures. Runtime environments like Sun's Java Virtual Machine (JVM) and Microsoft's Common Language Runtime (CLR) have broken ground in this direction and taken advantage of available hardware resources to provide practical abstracted execution environments for the Java and C# languages, respectively. We advocate and demonstrate the use of lightweight binary supervision for C, C++, and `x86` programs. The ability to intercept, insulate, and otherwise supervise the execution of these programs (a large part of the legacy code base) can only improve the security of our critical infrastructure.

We believe that our repair policy language, Ripple, can affect how software is updated and tested. While fault injection is a traditional method of testing programs, Ripple makes fault injection particularly easy. When combined with STEM's lightweight binary supervision, Ripple can make controlled, rational fault injection a practical method of testing. The use of Ripple for both repair and fault injection reflects the versatility of the language, even though it was initially designed to be domain–specific for self–healing. This original "repair" use most clearly has implications for dynamic software updating. Ripple provides a non–invasive method of software system control that is fundamentally different than applying arbitrary and oftentimes obfuscated binary patches. Of course, challenges such as computing Ripple deltas and side effects still remain, but Ripple's other benefits (such as being easy to turn off if the Ripple policy is found to negatively impact the software system — something that cannot be done with binary patches like Windows Update) may outweigh such considerations.

1.7 Organization

The overall flow of this thesis proceeds from an examination of related work and a brief review (at a medium level of detail) of some of the underlying principles and foundations of integrity postures to a detailed examination of the key contributions and an evaluation of their performance and efficacy. The latter half of the thesis discusses three major ideas: the design and implementation of the supervision environment (a way of executing the

ROAR workflow), the repair policy model and language, and the exploit filter generation and enforcement environment.

Chapter 1 has, until this point, described to the reader *what* we do to address the central problem of software self–defense. The remainder of the thesis informs the reader *how* we will go about it.

Chapter 2 discusses a range of related work in the space of systems security. It focuses mainly on how researchers have tried to make software more robust against a variety of attacks. It also reviews recent work in the space of exploit and vulnerability signature generation, as well as the emerging area of software self–healing. It includes a brief review of ASSURE, one of the most closely related systems to the work presented in this thesis.

Chapter 3 presents very specific background material that the reader may require in order to understand and contextualize the core mechanisms and techniques used by the system presented in this thesis. This material is too detailed to fit naturally into the problem introduction, and much too specific to the thesis ideas to be relegated to Chapter 2. This chapter provides more detail on the ROAR workflow, microspeculation, the foundations of our integrity repair model, background on constraint satisfaction problems, and an overview and evaluation of the technique of error virtualization [SLBK05].

Chapter 4 contains part of the heart of the thesis. It discusses in detail the implementation of two versions of our runtime supervision framework, STEM. STEM provides a key service: a microspeculated execution pipeline. This chapter examines two implementations of STEM, one based on software emulation, the other based on binary rewriting using Intel's Pin [LCM+05] binary rewriting framework. A discussion of how both versions of STEM interoperate with FLIPS is provided in Chapter 6.

Chapter 5 contains the second core part of the novel work in this thesis. It discusses in detail our integrity repair model and its application to maintaining an internal integrity posture for a software system. This chapter also contains a review of our repair policy language, Ripple. More detail on Ripple is available in Appendix A.

Chapter 6 completes the trio of novel contributions at the heart of this thesis. It relates the design and implementation of FLIPS, our novel exploit signature generation system and its integration with both versions of our host–based supervision framework, STEM. We also

evaluate the speed and power of exploit filter generation and enforcement.

Chapter 7 presents and evaluates the use of behavior modeling based on function return values. Since return values form an integral part of our self–healing mechanism, we explore what kind of modeling can be accomplished using them. In particular, profiling the behavior of applications (especially when the application is a black box or in situations where we lack source code access) can assist efforts to *detect* deviations from normal operation, *repair* aspects of the data and control flow, and *validate* those repairs.

Chapter 8 provides an evaluation of the systems and material presented in the previous chapters. In particular, we assess the performance impact of both implementation versions of microspeculation, the coverage tradeoff of selective supervision, how effective repair policies are for real vulnerabilities, and the speed of repair.

Chapter 9 is somewhat more open–ended in nature. It covers some of the fundamental limitations of the work in this thesis, such as the challenge of Automatic Repair Validation and the need to extend the implementations to perform kernel–level supervision (something which existing technologies like a VMM (*e.g.*, Xen), a binary-rewriting virtualization environment like PinOS, or a binary-rewriting based whole–system emulator like QEMU can enable). It also looks forward to some research opportunities for applying the work presented in this thesis, including the use of the framework as part of a collaborative security environment [LPKS05, WLK06] such as an Application Community [LSK06a, LSK05, LSK06b]. We close this chapter with a brief contemplation of some of the ethical considerations of self–healing software and software self–defense.

Chapter 10 summarizes the contributions of the thesis and our major results, and describes how they help address the problems presented above.

Chapter 2

Related Work

Defending software applications against various forms of attack has been a focus of the systems security research community for some time, and a wide variety of techniques have been proposed to harden systems during development (proactive) or at runtime (reactive). In addition, as traditional anti-virus measures prove ineffective against the vast array of new malware, a number of detection techniques attempt to alert systems to the occurrence of never-before-seen exploit input (*i.e.,* zero-day attacks) or the discovery of previously unknown vulnerabilities. Detecting such attacks and vulnerabilities can make use of work in both host–based and network–based anomaly detection, as well as more invasive host instrumentation aimed at extracting precise information about the exploit or vulnerability in order to prevent its recurrence.

A principal focus of this thesis is on augmenting detection systems with adaptive response mechanisms to preserve both the internal and external integrity of the protected or supervised software application. This work is a part of the emerging area of research on self–healing software systems. Automated response systems vary from the low–tech (manually shut down misbehaving machines) to the highly ambitious (on the fly "vaccination", validation, and replacement of infected software). In the middle lies a wide variety of practical techniques, promising technology, and nascent research.

2.1 Traditional Defense Mechanisms

Attempting to remove flaws and vulnerabilities from code before it is deployed is standard software engineering practice, and a number of methodologies, test frameworks, profiling tools, and static analysis source code scanners exist. Unfortunately, software engineers and testers, no matter how exhaustive their efforts, are unable to remove all extant bugs for a variety of reasons. Systems, therefore, will always contain errors.

Although one major thrust of this thesis is to advocate the adoption of self–healing techniques, we do so primarily as a way to alleviate the risk associated with the discovery and exercise of these errors by an attacker rather than a bulletproof method of exception handling (in fact, Chapter 5 explicitly considers the differences between self–healing and traditional exception handling). We do not argue for the wholesale abandonment of prerelease or offline vulnerability identification. Replacing ongoing efforts to identify vulnerabilities throughout the lifecycle of a piece of software with a foolhardy reliance on any single component — self–healing or otherwise — is a recipe for disaster and violates the fundamental security tenet of "defense-in-depth." Nevertheless, we consider most such offline methods out of the scope of the related work of this thesis.

Proactive defense mechanisms vary widely in range and scope. Options include writing the system in a type-safe language that respects and monitors memory object boundaries (*e.g.,* Java) or using "safer" versions of standard C library functions [BST00]. Other approaches involve employing artificial diversity of the address space [BDS03, SPP+04] or instruction set [KKP03, BAF+03, BK04], along with compiler–added integrity checking of the stack [CPM+98, Eto00] or heap variables [SGK05]. Other systems explore the use of tainted dataflow analysis to prevent the use of untrusted network or file input [CCCR05, NS05] as part of the instruction stream. The contributions of this thesis are motivated in part by Sidiroglou *et al.*'s [SLBK05] concept of error virtualization and software vaccination [SK03].

Network defense systems (typically composed of IDSs and firewalls) have shortcomings that make it difficult for them to identify and characterize new attacks and respond intelligently to them. In particular, opportunistic encryption, high traffic rates at or near the network core, and a non-endpoint view of traffic all make it difficult for the IDS or firewall to know how traffic is processed at the end host [HPK01]. These obstacles motivate

the argument for placing protection mechanisms closer to the end host (*e.g.*, distributed firewalls [IKBS00]). This approach to system security can benefit not only enterprise-level networks, but home users as well. The principle of "defense-in-depth" suggests that traditional perimeter defenses be augmented with host-based protection mechanisms. This thesis advocates one such system to adaptively react to new exploits.

2.1.1 Protecting Control Flow

Work in assuring that the execution of a process does not go astray has typically focused on protecting the integrity of a process's machine–level jump or branch target memory addresses. For example, StackGuard and related approaches [CPM+98, Eto00] attempt to detect changes to the return address (or surrounding data items) of a stack frame or activation record. If the integrity of these values is violated, execution halts.

Starting with the technique of *program shepherding* [KBA02], the idea of enforcing the integrity of a process's control flow has been increasingly investigated. Program shepherding transparently validates branch instructions in IA-32 binaries for Linux and Windows to prevent transfer of control to injected code, and to make sure that calls into native libraries originate from valid sources. This approach is implemented on the RIO [DA02] architecture; the system stores the decision (along with translated code) in a cache, thus incurring little overhead.

Abadi *et al.* [ABEL05] propose formalizing the concept of Control Flow Integrity (CFI), observing that high–level programming often assumes properties of control flow that are not enforced at the machine language level. CFI provides a way to statically verify that execution proceeds within a given control flow graph (the CFG effectively serves as a policy). The use of CFI enables the efficient implementation of a software shadow call stack with strong protection guarantees. CFI complements our approach in that it can enforce the invocation of our repair procedures (rather than having malcode attempt to skip past these checks). The integrity model we suggest encompasses both data and control flow.

Control flow is often corrupted because an unconstrained data item enters the system and is eventually incorporated into part of an instruction's opcode, set as a jump target, or forms part of an argument to a sensitive system call. A great deal of work has focused on dataflow

analysis of tainted data and ways to prevent such attacks from succeeding [SLZD04, NS05, NBS06, CCCR05]. Bhatkar, Chaturvedi and Sekar's [BCS06] work on dataflow anomaly detection examines the temporal properties of data flow through a sequence of system calls in order to detect intrusion attempts. One benefit of this model of dataflow integrity is that temporal properties enable formal reasoning about the security properties of a program.

2.1.2 Artificial Diversity

Software monoculture has been identified as a major problem for networked computing environments [Got03, Gee03, Sta04]. Monocultures act as force amplifiers for attackers, allowing them to exploit the same vulnerability across thousands or millions of instances of the same application. Such attacks have the potential to rapidly cause widespread disruption, as evidenced by several incidents over the past few years.

A number of efforts have been made to protect software monocultures via the introduction of artificial diversity [RJCM03, CS02, OS04], either in a manual (*i.e.,* n-version programming teams) or automated fashion. Creating a large number of different systems manually [Avi85], however, not only presents certain practical challenges, but can result in systems that are not diverse enough [BKL90].

As a result, research has focused on creating artificial diversity by introducing "controlled uncertainty" into system parameters that the attacker must govern in order to carry out a successful attack. Such parameters include the instruction set [KKP03, BAF$^+$03], the high-level implementation [RJCM03], the execution language [BK04], the memory layout [BDS03], and the operating system interface [CS02], with varying levels of success [SPP$^+$04]. Holland, Lim, and Seltzer [HLS04] introduce the idea of automatically generating randomized architectures. Since synthesizing the hardware for every such generated architecture is an untenable approach, they recommend using VMMs to provide the necessary execution environments. Diversity, however, creates its own set of problems involving configuration, management, and certification of each new platform [Whi03]. In certain cases, such environments can decrease the overall security of the network [Pre99].

2.1.3 Execution Supervision Environments

Virtual machine emulation of operating systems or processor architectures to provide a sandboxed environment is an active area of research. Virtual machine monitors (VMMs) are employed in a number of security–related contexts, from autonomic patching of vulnerabilities [SK03] to intrusion detection [GR03]. MiSFIT [SS98] is a tool that constructs a sandbox by instrumenting applications at the assembly language level.

Lee *et al.* [LKMS03] propose a hardware–based return stack (SRAS) to frustrate buffer overflow attacks. Suh *et al.* [SLZD04] propose hardware extensions to thwart control transfer attacks by tracking "tainted" input data (as identified by the OS). If the processor detects the use of this tainted data as a jump address or an executed instruction, it raises an exception. Kuperman *et al.* [KBO+05] provide an overview of buffer-overflow related attacks and discuss some hardware-based approaches to protection, including SRAS (and related variants) and their own SmashGuard proposal.

The Copilot system [PFMA] by Petroni *et al.* is one expression of hardware security aimed at integrity protection. Much like the Tripwire[1] software, the goal of Copilot is to make sure that important data has not been corrupted. Copilot performs rootkit intrusion detection by monitoring changes to a host's kernel text segment and related data structures. The current implementation is based on a PCI card that monitors the host's main memory via DMA (without the host kernel's knowledge) and has a secure communications link to an administrative reporting station.

The work by Dunlap, King, Cinar, Basrai, and Chen [DKC+02] is closely related to some of the work presented in this thesis. ReVirt is a system implemented in a VMM that logs detailed execution information. This detailed execution trace includes non–deterministic events such as timer interrupt information and user input. ReVirt's smaller codebase (relative to a full-blown OS) should inspire more confidence in its correctness and level of resistance to attack or subversion. ReVirt's primary use, however, is as a forensic tool to replay the events of an attack, while the goal of this thesis is to provide a mechanism for protecting code against malicious input *at runtime*.

[1] `http://tripwire.org/`

An interesting application of ReVirt [DKC+02] is BackTracker [KC03], a tool that helps identify the steps involved in an intrusion post mortem. Because detailed execution information is logged, a dependency graph can be constructed backward from the detection point to provide forensic information about an attack. Malfor [NZ06] is another system that tries to identify the processes involved in an attack/infection through automated experiments in which the system configuration is changed slightly each time, to identify the importance of each process to the attack.

2.2 Automated Defense

Existing approaches to software system defense stop attacks from succeeding by preventing the injection of code, transfer of control to injected code, or misuse of existing code. Most of these defense mechanisms, however, usually respond to an attack by terminating the attacked process. Even though it is considered "safe", this approach is unappealing because it leaves systems susceptible to the original fault upon restart and risks losing accumulated state. Self–healing mechanisms complement these techniques by considering how to recover execution to a safe path, thereby maintaining availability.

The pH system [SF00] foreshadows the development of automatic reaction systems. Its aim is to frustrate an attacker by using system call interposition to distinguish "self" from "non-self" and slow down an attacker's code. While not strictly self–healing (in that it does not correct the underlying fault), this system was among the first to propose an active reaction mechanism (feedback to delay system call execution) to foil attacks and is representative of the seminal work in artificial immune systems. The pH system walks a fine line between introducing unexpected or semantically incorrect behavior and frustrating an attacker's will.

2.2.1 Self–Healing

Software self–healing is an active area of research. Unfortunately, self–healing is a somewhat overloaded term, and it has traditionally been used to describe distributed systems consisting of many identical or near identical components that can transparently fail over to

a backup or spare replica component. We are primarily concerned with protecting single instances of heterogeneous systems like common COTS (Commercial Off The Shelf) software, including web browsers, word processors, web and data servers, and media players.

Systems can self–heal in a variety of ways. Some first efforts at providing effective remediation strategies include failure oblivious computing [RCD+04a], error virtualization [SLBK05], rollback of memory updates [SC05], crash-only software [CF03], and data structure repair [DR03]. The first two approaches (explained in more detail below) may cause a semantically incorrect continuation of execution (although the Rx system [QTSZ05] attempts to address this difficulty by exploring semantically safe alterations of the program's environment). The ability to rollback [BP02] and cleanly restart [CF03] is important to self–healing systems. Crash-only software employs a technique of micro-rebooting (*i.e.*, restarting individual system components) in case a failure occurs. In this type of software, crashing is the default method of restart, so it is gracefully handled. Unfortunately, the approach requires that software systems be completely rewritten to adopt the model and data snapshot services required to safely micro-reboot.

One of the most critical concerns with recovering from software faults and exploited vulnerabilities is ensuring the consistency and correctness of program data and state. An important contribution in this area is the work by Demsky *et al.* [DR03], which discusses mechanisms for detecting corrupted data structures and fixing them to match some prespecified constraints. While the precision of the fixes with respect to the semantics of the program is not guaranteed, their test cases continued to operate when faults were randomly injected. The notion of data structure repair is closely related to our approach and serves as partial inspiration for the idea.

Rinard *et al.* [RCD+04b] have developed compiler extensions that deal with access to unallocated memory by automatically expanding the target buffer (in the case of writes) or manufacturing a value (in the case of reads). Such a capability aims toward the same goal as this thesis: rather than simply crashing, provide a more robust fault response. The authors leverage this technique and introduce *failure–oblivious computing* [RCD+04a], an idea that slightly predates and is related to our *error virtualization* hypothesis [SLBK05].

As with all systems that rely on rewinding execution [BP02, QTSZ05] after a fault has

been detected, I/O with external entities during self–healing remains uncontrolled. For example, if a server program supervised by a self–healing mechanism writes a message to a network client during supervision, there is no way to "take back" the message: the state of the remote client has been irrevocably altered.

ASSURE [SLKN07] is a novel attempt to minimize the likelihood of a semantically incorrect response to a fault or attack. ASSURE proposes the notion of *error virtualization rescue points*. A rescue point is a program location that is known — or at least conjectured — to successfully propagate errors and recover execution. The key insight is that a program will respond to malformed input differently than legal input; during an offline training phase, the system learns about locations in the code that successfully handle these sorts of anticipated input "faults." These locations serve as good candidates for recovering to a safe execution flow. This technique is combined with operating system virtualization [OSSN02] to help ensure that recovery actions do not interfere with the normal operation of the system. ASSURE can be understood as a type of structured exception handling that dynamically identifies the best scope to handle an error (rather than statically determined by a programmer). In terms of self–healing systems, we view the approach taken by this thesis and the approach taken by ASSURE as complementary: we seek the best data integrity policies to use *at the function that fails or behaves in an abnormal way;* ASSURE seeks *the best location* to which to "teleport" a failure that is detected elsewhere in the program.

Rx [QTSZ05] checkpoints execution in anticipation of errors. When an error is encountered, execution is rolled back and replayed with the process's environment changed in a manner consistent with the semantics of the APIs the code uses — a clever attempt to avoid the semantically incorrect responses of previous systems [RCD+04a, SLBK05]. This procedure is repeated with different environment alterations until execution proceeds past the detected error point. The system reverts to crashing if no safe alteration can be found.

Modeling executing software as a transaction that can be aborted has been examined in the context of language-based runtime systems (namely, Java) [RW02, RW01]. That work, focused on safely terminating misbehaving threads, introduces the concept of "soft termination." Soft termination allows thread termination while preserving the stability of the language runtime. In that approach, threads (or *codelets*) are each executed in self-

encompassing transactions, applying standard ACID semantics. This allows changes to the runtime's (and other threads') state made by the terminated codelet to be rolled back. The performance overhead of that system can range from 200% up to 2300%.

Oplinger and Lam propose [OL02] using hardware Thread-Level Speculation (TLS) to improve software reliability. Their key idea is to execute an application's monitoring code in parallel with the primary computation and roll back the computation "transaction" depending on the results of the monitoring code. Candea and Fox propose a different approach: design software systems such that they employ crashing as the normal halting mode and use recursive micro-reboots to safely restart [CF03].

2.2.2 Survivable Systems

Traditional fault-tolerance techniques are a related area of work, although they are primarily intended to supply enough resources for a particular enclave to survive an attack by outlasting the resources of an attacker. This thesis does not focus on providing Byzantine robustness for a collection of network nodes, although the techniques we propose may be useful, for example, in an Application Community [LSK06a]. Instead, this thesis focuses on giving enough time and space within a single process or group of related processes for them to deal with previously unseen exploits and previously undiscovered vulnerabilities.

Secure survivable architectures are typically very application- or domain-specific. Ghosh *et al.* [GV99] propose "fault injection analysis" applied to software. Strunk *et al.* [SGP+02] apply a low-level approach: they propose an intrusion detection and recovery model at the storage layer. Kreidl *et al.* [KF02] propose a formalized feedback-driven model for individual COTS applications. SABER [KPG+03] is a generalized, application-neutral architecture that encompasses a broad array of tools. The APOD project [ati03] uses a combination of intrusion detection, firewalls, TCP stack probes, virtual private networks, bandwidth reservation, and traffic shaping mechanisms, to allow applications to detect attacks and contain the damage of successful intrusions by changing their behavior. They also discuss the use of randomizing techniques, such as changing the TCP ports applications listen to.

2.3 Anomaly Detection

Anomaly-based classification is a powerful method of detecting inputs and behavior (including network traffic content [WPS06, WS04, KTK02] and sequences of system calls [CC02, SF00]) that are probably malicious without relying on a static set of signatures. PAYL [WCS05] models the 1-gram distributions of normal traffic using the Mahalanobis distance as a metric to gauge the normality of incoming packets. Anagram [WPS06] caches known benign n-grams extracted from normal content in a fast hash map and compares ratios of seen and unseen grams to determine normality.

The perceived utility of anomaly detection is based on the assumption that malicious inputs are rare in the normal operation of the system. However, since a system can evolve over time, it is also likely that new *non-malicious* inputs will be seen [FSA97, SF00]. Perhaps more troubling, Fogla and Lee [FL06] have shown that it is possible to evade anomaly classifiers by constructing polymorphic exploits that blend in with normal traffic (a sophisticated form of mimicry attack [WS02]), and Song *et al.* [SLS+07] have improved on this technique and shown that content–based approaches may not work against all polymorphic threats[2], since the malcode analysis is often fixed on specific byte patterns [NKS05]. Finally, Taylor and Gates have suggested that anomaly detection (as originally conceived for host-based monitoring) is an ill fit for current network traffic [TG06].

The seminal work of Hofmeyr, Somayaji, and Forrest [HSF98, SF00] helped initiate application behavior profiling at the system call level. Most approaches to host-based intrusion detection perform anomaly detection [CC02, Pro03, GRS04, GDJ+05] on sequences of system calls because the system call interface represents the services that malcode, once activated, must use to effect persistent state changes and other forms of I/O. Such system call level information is easy to collect; the `strace` and `ltrace` tools for various flavors of Unix are built to do exactly this task.

The work of Feng *et al.* [FKF+03] includes an excellent overview of the literature circa 2003. The work of Bhatkar *et al.* [BCS06] also contains a good overview of the more recent

[2]Randomized modeling — that is, randomizing the feature selection to introduce uncertainty into the attacker's perspective of "normal" may be a promising way forward to combat these attacks.

literature. Noting that most previous approaches measure control flow, they provide a technique for *dataflow* anomaly detection (Mutz *et al.* [MVVK06] also examine anomaly detection on system call arguments). Behavior profiling helps create policies for detection [Pro03, LcC04] and could potentially be used to automatically generate repair policies, as suggested by this thesis.

2.4 Exploit Signature Generation

Automatically generating exploit signatures has been the focus of a great deal of research. The early days of network signature generation often ignored content altogether, preferring instead to generate simple filters based on frequency analysis of traffic directed to certain ports (mainly to frustrate fast–spreading worms, the predominate threat at that time). Both Autograph [KK04] and Earlybird [SEVS04] use traffic characteristics (*e.g.*, frequency of various packet types) to generate worm signatures. Other approaches perform more sophisticated statistical analysis [NKS05, WS04, WCS05, YGBJ05].

Perhaps more interesting is a series of tools aimed at identifying binary malcode inside network packets rather than generating a traditional signature [WPLZ06, TK02, SCM04, PAM06]. Some of these approaches rely on identifying a NOP sled (the sequence of instructions in an exploit whose purpose is to guide the program counter to the exploit code). Others attempt to detect polymorphic code by learning a control flow graph for the malcode [CB05, KKM+05].

Finally, a large number of schemes propose capturing a representation of the exploit to create a signature for use in detecting and filtering future versions of the attack. These host–based approaches divert traffic through an instrumented version of the application to detect malcode. If confirmed, the malcode is dissected to dynamically generate a signature to stop similar future attacks [LS05, XNK+05, LWKS05]. Liang and Sekar [LS05] and Xu *et al.* [XNK+05] concurrently propose using address space randomization to drive the detection of memory corruption vulnerabilities and create a signature to block further exploits. Our FLIPS system [LWKS05], described more fully in Chapter 6, uses an anomaly sensor, filtering proxy, and self–healing supervision framework to detect, recover from, and block

attacks. The system proposed by Anagnostakis *et al.* [ASA+05] has many of the same goals as FLIPS. There are a number of differences in architecture and implementation. In particular, the system uses an anomaly sensor as a traffic classifier to split traffic processing into two service paths: one path for requests that are deemed normal and the other for requests the AD sensor believes are abnormal. FLIPS contains a single service path and does not rely on the AD sensor's decision to accept or reject a particular network message. In addition, by using STEM as a supervision environment, we enable FLIPS to detect and prevent all instances of code injection attacks, not just stack–based buffer overflows. Finally, FLIPS is designed to protect a host without the need for a shadow or replica.

2.5 Vulnerability Signatures

Recent work [WGSZ04, SLS+07] calls into question the ultimate utility of exploit–based signatures, and research on *vulnerability–specific* protection techniques [CSWC05, BNS+06, JKDC05] explores methods for defeating exploits despite differences between instances of their encoded form. In this Section, we only consider vulnerability signatures generated dynamically at runtime for deployed systems, rather than systems that attempt to find vulnerabilities statically during compilation or semi-statically through symbolic execution, as is done by EXE and DART (Directed Automated Random Testing) [CGP+06]. Identifying the underlying vulnerability may help reason about the appropriate action to take to heal or repair an attacked process.

2.5.1 Capturing Vulnerabilities

The underlying idea relies on capturing the characteristics of the vulnerability. Researchers in this area have represented one primary characteristic as a conjunction of equivalence relations on the set of branch target addresses that lead to the exercise of the vulnerability itself. Brumley *et al.* [BNS+06] supply an initial exploration of some of the theoretical foundations of *vulnerability–based signatures*. Vulnerability signatures help classify an entire set of exploit inputs rather than a particular exploit instance.

As an illustration of the difficulty of creating vulnerability signatures, Crandall *et*

al. [CSWC05] discuss generating high quality vulnerability signatures via an empirical study of the behavior of polymorphic and metamorphic malcode[3]. The authors present a vulnerability model that explicitly considers that malcode can be arbitrarily mutated. They outline the difficulty of identifying enough features of an exploit to generalize about a specific vulnerability. The critical features of an exploit may only exist in a few or relatively small number of input tokens, and if the attacked application is using a binary protocol, telltale byte values indicating an attack may be common or otherwise ordinary values. For example, the Slammer exploit essentially contains a single "flag" value of `0x4`. For other protocols, detecting that the exploit contained the string "HTTP" or some URL typically does not provide enough evidence to begin blocking arbitrary requests — or if it does, Song's analysis [SLS+07] indicates that such exploits can be arbitrarily mutated, thus vastly increasing the signature database and the processing time for benign traffic.

2.5.2 Systems

Vigilante [CCCR05] is a system motivated by the need to contain rapidly spreading malcode, such as Internet worms. Vigilante supplies a mechanism to detect an exploited vulnerability (by analyzing the control flow path taken by executing injected code). While Vigilante does not address the self–healing or recovery of a piece of exploited software, it does define an architecture for production and verification of Self-Certifying Alerts (SCA's): a data structure for exchanging information about the discovered vulnerability. A major advantage of this vulnerability–focused approach is that Vigilante can be agnostic to a particular exploit and can potentially be used to defend against polymorphic worms.

[3]Some researchers make no distinction between these types of malcode. We consider them distinct. Polymorphic malcode performs a straightforward masking operation (*e.g.*, encryption, XOR encoding, compression) or otherwise simple content transformation. Metamorphic malcode changes the actual instructions involved. Metamorphic malcode can change the attack vector that it uses and can change the syntax of the exploit it carries without significantly changing the semantics. For this reason, metamorphic malcode typically does not require a decoding phase or an embedded decoder. On the other hand, polymorphic code uses an encoding procedure to change its own appearance. The resulting sample consists of a decoder and the remainder of the malcode, which may look like completely random data *or* data that is indistinguishable from "normal."

Newsome and Song [NS05] propose dynamic taint analysis to monitor for injection attacks. The technique is implemented as an extension to Valgrind [NS03] and does not require any modifications of the program's source code. One way to minimize the hefty cost of complete binary supervision is to rewrite the program binary in response to an attack against a fully instrumented instance of an application, inserting only the necessary instrumentation for keeping track of taint information to detect that specific attack. The authors extend the system to do so [NBS06] with vulnerability–specific execution filters (VSEF), an idea with a different mechanism, but similar goals to Shield [WGSZ04]. VSEF's central mechanism of identifying a program control flow path that has been exploited is similar to one of Vigilante's solutions to the same problem.

Systems like Vigilante and VSEF may identify only a single control flow path rather than all possible paths to a particular vulnerable state (and thus not the entire "vulnerability"). As a brief example, a single control flow path at the binary level represents only a finite number of executed instructions; if this path includes a loop (common for vulnerabilities that iterate over input), the execution of the loop with one more or one less iteration would result in a slightly different control flow path. Another example at a somewhat higher level of abstraction is a vulnerability that is present in a common library function: there are potentially many control flow paths that enter this function. In some cases, these shortcomings make it easy for an attacker to evade such "vulnerability–specific" defenses (for example, by including an extra character in the exploit) while still triggering the same vulnerability. Vulnerability filters that are overly specific or constrained (such as ones that believe a loop must be *exactly* k iterations in length) may block too narrow a set of exploits.

Two research efforts attempt to capture a broader notion of the set of program paths related to a single vulnerability. The first, ShieldGen [CPWL07], does so by associating a single sample control flow with the protocol or file format parsing states of an external network filtering component. ShieldGen combines tainted dataflow analysis (similar to that used in the Vigilante system [CCCR05]) and protocol or data format parsing [BBW+07] to construct "data patches" to filter input related to a particular vulnerability.

The second, Bouncer [CCZ+07], performs program slicing to reliably identify the actual set of program paths. Bouncer helps revive the notion that content filtering is a viable way

to defend systems even in the presence of polymorphic exploit input. Bouncer (similar to ShieldGen) captures a sample exploit instance and the effect it has on an instrumented application (namely, the binary control flow path that the exploit input causes the application to take). Bouncer then enters a feedback loop intended to identify other control flow paths and, consequently, other conditions on input that help identify other "legal" exploit forms.

While ShieldGen also uses a feedback loop, the purpose of ShieldGen's feedback loop is to use a protocol parsing engine to generate other inputs guided by the protocol grammar. In contrast, Bouncer performs program slicing to identify actual control flow paths in the application. This approach is more robust, as there is no guarantee that the published protocol specification matches the actual application code. While no solution can be simultaneously sound and complete, it is not clear that ShieldGen provably provides either property. Bouncer filters are sound; it remains unclear at the time of writing, however, if Bouncer extensively covers the space of possible exploit inputs in a reasonable amount of time, as program slicing is a potentially expensive form of analysis.

Bouncer offers a realistic way forward for intrusion defense; it walks a fine line between simple content–based exploit filters and more drastic self–healing approaches. The former can be defeated by almost trivial polymorphic techniques, as we show in other work [SLS+07], and the latter have a major problem sustaining application semantics (a problem that this thesis, in part, addresses). Thus, system defenders are faced with a bit of quandary: they would like to use content filtering (it has fairly cheap runtime overhead, can protect more than a single application at a time, and is easy to implement), but it is almost trivial for a smart and determined attacker to bypass. On the other hand, although more invasive (and complicated) defense mechanisms can defeat a family or class of exploits, these mechanisms often cause a self–induced DoS (*i.e.,* application crash) at best.

Blocking malicious input in an *intelligent* way — that is, in a way that closely matches the application's actual behavior — helps filtering become agnostic to minor variations in the exploit input *without* unnecessarily disturbing the semantics of application execution. The intuition is compelling: if "bad" input never reaches an application, we need not unduly clutter an application's execution with complicated defense mechanisms.

2.6 Other Work

Countering attacks and malcode is a hard problem. Spinellis showed that identification of bounded length metamorphic virii is NP-complete [Spi03] by decomposing the problem into one of graph isomorphism. In addition, Fogla *et al.* [FL06] show that finding a polymorphic blending attack is also an NP-complete problem.

Gamma [OLHL02, BOH02] is an architecture for instrumenting software such that information that can lead to future improvements of the code can be gathered in a central location, without imposing excessive overhead to any given code instance. Their technique, software tomography, is similar to our code-slicing approach, and has been combined with a dynamic software update mechanism that allows code producers to fix bugs as they are detected. Our work is different primarily in that (*a*) we introduce *a fully automated* mechanism for software healing, (*b*) which does not require merging of the monitoring information from the different software instances.

The Cooperative Bug Isolation project [LAZJ03, LNZ+] uses a sampling infrastructure to gather information from a program's execution and communicates its findings to a central database where the data is analyzed to extract debugging information automatically. In order to reduce the instrumentation cost they statistically spread the monitoring across an application and a large user base. This approach is similar to our work on Application Communities [LSK06a, LSK05].

Since security is often delegated to a secondary concern in the production of software systems, many researchers have undertaken work that attempts to retrofit the enforcement of security policies on these codebases. In effect, our work on supplying a repair policy attempts to retrofit an exception handling mechanism on code that does not already contain one. In this sense, the work by Ganapathy *et al.* [GJJ06] is most closely related to our work on repair policy. They discuss ways to retroactively insert authorization checks into the X server. They face many of the same issues that we do, including identification and placement of calls to the policy evaluator or reference monitor and specifying the content of the policy. Our work differs in two important aspects. First, we focus on integrity policy and how to automatically repair violations of that policy. Second, our system works by dynamically instrumenting programs binaries and does not require source code to operate (although

more precise policies can be constructed with knowledge of source-level names and objects).

Chapter 3

Foundations of Integrity Postures

In this chapter, we discuss the background knowledge necessary to understand the concepts underlying integrity postures and our solution to the problem of software self-defense. The security property of integrity is fundamental to our solution, but it is important to note a key distinction between our work and traditional methods of integrity measurement and maintenance. Whereas systems like TripWire[1] focus on detecting violations to the integrity of program binaries or configuration data on disk, we are primarily concerned with *restoring* the integrity of a running program during execution should its integrity be violated. Maintaining an execution integrity posture is not equivalent to program shepherding [KBA02] or control flow integrity [ABEL05]; we are not primarily concerned with the value of the instruction pointer *per se*. Although the core issue of detecting and preventing modifications to the flow of instructions remains the same, the instruction pointer is but one important execution artifact and data item that is subject to an integrity repair policy.

This chapter places the core contributions of this thesis, as related in Chapters 4, 5, and 6, in context. While Chapter 2 discussed related work in the general space of software defense, this chapter recounts four specific ideas that provide the underlying theoretical foundation for integrity postures: the ROAR workflow (a novel contribution of this thesis), policy–constrained microspeculation (also a novel contribution of this thesis), the Clark-Wilson Integrity Model (novel modifications of which we discuss in Chapter 5), and the

[1] `http://www.tripwire.org`

basics of constraint satisfaction (including a brief review of RealPaver, the constraint solver we use as part of our software self-defense system).

We combine each of these ideas to help frame a solution to the problem of software self-defense. In order to maintain an integrity posture, software must execute some supervision workflow. The stages of that workflow help define the operations of a speculative execution pipeline controlled by a repair policy. The effects of speculated instructions that do not maintain execution integrity are corrected according to automated satisfaction of the constraints expressed in the repair policy.

We close the chapter with a discussion and evaluation of the basic error virtualization hypothesis. Part of the motivation of the work in this thesis is derived from the use of the technique of error virtualization as a self–healing mechanism.

3.1 The ROAR Workflow

The ROAR (Recognize, Orient, Adapt, Respond) workflow imposes a logical organization on attempts to specify a self–healing process for software systems. We note that symptoms of attacks and symptoms of more mundane system faults can often appear similar. This work is primarily concerned with addressing symptoms of attacks. In the logical extreme, software self–healing can be mostly agnostic to whether the cause of an incident is either a software fault or an exercised vulnerability. We do not examine how to deal with non-malicious software faults, although we believe most of the techniques could be applied equally well.

The workflow consists of four stages that we consider necessary to enact a self–healing repair. Systems that employ ROAR first (*a*) *Recognize* a threat exists or an attack has occurred, then (*b*) *Orient* the system to this threat by analyzing it, next (*c*) *Adapt* to the threat by constructing appropriate fixes or changes in state, and finally (*d*) *Respond* to the threat by verifying and deploying those adaptations. This workflow is a novel contribution of this thesis. The closest concept to ROAR is the OODA (Observe, Orient, Decide, Act) feedback loop, and the key distinction is one of specificity; OODA is a general decision–making process created in the context of military strategic and tactical combat decisions,

whereas ROAR speaks specifically to the critical stages of a self-healing workflow. The ROAR workflow may be mapped to OODA if appropriate for the problem domain.

3.1.1 Stage 1: Recognition and Detection

Attacks (more precisely, symptoms of attacks) must first be detected before a response can be mounted. A system must recognize that the integrity of its execution has been corrupted. This thesis relies on two types of detectors: detection mechanisms that have been proposed, researched, and implemented by third parties (and re-implemented as part of our self–healing framework) and detection mechanisms that are specified as constraints within the repair policy itself. Although most of our "black-box" detectors are geared toward detecting binary code injection, we have built a detector for sequences of anomalous function calls (more details of which are given in Chapter 9), and our approach is agnostic to the particular type of attack or failure as long as a reliable detector for the attack conditions or symptoms can be written. In the cases where this is possible, we can plug the detector into our framework as a sensor.

3.1.2 Stage 2: Orientation and Diagnosis

Once attack symptoms are detected (*e.g.,* an overwrite of the stack-resident return address), the system must orient itself to the attack by analyzing what else the attack has corrupted and determining its origin. This diagnosis step is crucial[2]. In our system, STEM, we perform an online forensics step to correlate the data of a code injection attack with the input to the system. Doing so allows us to generate accurate exploit filters. Diagnosis also occurs during repair policy evaluation, in which a series of constraint relationships are assessed to determine the set of data items requiring repair.

3.1.3 Stage 3: Adaptation and Repair

After diagnosing the symptoms of an attack or failure, the adaptation stage of ROAR forms the heart of a self–healing response. The goal of this stage is to enable the system to (at

[2]In Vigilante [CCCR05], a subsystem is dedicated to the replay of the suspected attack input to confirm that it causes a server compromise and was not a false positive.

the very least) repair or undo the damage done during the attack or fault. Moreover, this stage provides the opportunity to strengthen the system against future attacks, a capability that some traditional biological examples of self–healing might not have. To wit, a cut to the skin results in repairing the skin tissue, not in skin that is "cut-proof."[3] Where possible, however, a repair of a system should make the system more robust in that it becomes "cut-proof" or resistant to a recurrence of the same fault or exploit instance. This type of behavior is much more akin to the properties of the vertebrate immune system, where antibodies for a particular type of attack are produced and remain resident to ward off future infections. The system, having been attacked, is stronger than it was before the attack. In any event, the result of this stage should include a configuration setting which repairs the damage caused by the attack and (optionally) a filter that limits the cause of the fault in the future. Such a filter may be either an input filter (such as an exploit signature) or a behavior filter (such as a vulnerability signature).

3.1.4 Stage 4: Response and Deployment

The final stage of ROAR involves the deployment of the fix, patch, or filter generated during the third stage. Deployment can be a non-trivial exercise. Interrupting system execution risks losing accumulated state, and repeatedly rebooting the system simply represents another form of self–induced DoS, although redesigning systems so that recursive micro-reboots become the standard of operation is an option [CF03] (we also advocate writing future applications in such a way as to be amenable to automated repair). More preferable is a mode where the fix, patch, or filter generated in the third stage is automatically and transparently inserted into the execution stream without a noticeable interrupt in processing by clients, peers, other communication partners, or users.

[3]It could be noted, however, that scar tissue can be somewhat tougher than the skin tissue it replaces, and that other types of tissue (notably bone) knit back together stronger than what was broken. Callouses resist future damage. These exceptions reveal the limitations of the self–healing immune system analogy.

3.2 Microspeculation

Portions of an application can be treated as a transaction. Microspeculation is the process of speculatively executing such arbitrary "slices" of a program by freely switching between supervised execution and native execution. A program slice could range from a single instruction to the entire program and anywhere in between: a single basic block[4], groups of basic blocks, a single source–level routine or a set of routines. Functions serve as a convenient abstraction and fit the transaction role well in many situations [SLBK05].

Microspeculation provides the time and space for a sensor to notice that a fault has occurred or a vulnerability has been exploited and to enact a repair to address this condition. In fact, microspeculation is the technique we use to support the execution of the ROAR workflow. The general aim of microspeculation is akin to systems [SK03, RJCM03, PF95] that utilize a secondary host machine as a sandbox or instrumented honeypot: work is offloaded to this host, thus minimizing exposure to the primary host. In our case, the execution of the primary instruction stream is delegated to an intermediate virtual supervision environment rather than the bare hardware.

Slicing at a very fine granularity (that is, supervising only a small subset of instructions, routines, or other units of execution) is done primarily as a performance consideration. Since program interception and supervision comes at a price in terms of performance, a careful balance between coverage of a program and slowdown must be achieved. Widespread coverage of "weak", suspect, or potentially vulnerable parts of a program helps detect faults and attacks. Larger monitored portions (in terms of cumulative slice size, measured in execution time or number of dynamic instructions) impose a concomitantly greater penalty.

If execution interception and supervision were free, then we could effectively cover or supervise the entire program for a very low cost, thereby providing our sensors the maximum opportunity to detect a fault or exploited vulnerability. More detail on experiments pertaining to the coverage tradeoff is given in Chapter 8.

[4] A basic block is a continuous straight-line sequence of machine instructions that does not contain a control flow transfer or branching instruction. Basic blocks typically only contain several machine instructions, often fewer than ten.

3.2.1 Speculative Execution

Speculative execution is a technique traditionally employed in hardware as a method of keeping a superscalar pipeline full. Program "slices", in this sense, are basic blocks (a limited length straight-line sequence of instructions). Speculative execution is used in microprocessors to execute the instructions in a code branch before the evaluation of the branch conditional is finished. The need to perform speculative execution arises in pipelined processors because the conditional instruction that the branch depends on has proceeded deeply into the pipeline but has not been evaluated by the time the processor is ready to fetch additional instructions. An example of this situation for a synthetic assembly language is shown in Figure 3.1.

While a complete discussion of the strategies for dealing with branch prediction is beyond the scope of this thesis, a basic overview of the subject and pointers to other material are available in Hennessy and Patterson's standard text [HP03]. Evers *et al.* [EPP98] investigate the predictability of branches and provide an overview of various branch prediction schemes that have been proposed to ameliorate the cost of incorrect predictions. Wang *et al.* [WFP03] explore an interesting result: about 50% of mispredicted branches do not affect correct program behavior. This result is encouraging because it offers evidence that our transactional technique of microspeculation holds at the machine level. Microspeculation and our use of it differs from standard speculative execution techniques by introducing an additional layer of speculative execution in which the acceptance of a particular execution path is not based on the evaluation of a branch conditional, but rather on a higher-order constraint dictated by an integrity repair policy.

3.2.2 Policy–Constrained Speculative Execution

Microspeculation can, in effect, provide a supervision environment with a glimpse of the future. This execution pipeline needs to be guided or constrained in some customizable way. Speculative execution of a program slice governed by a repair policy is a core contribution of this thesis. In doing so, we transform instruction–level speculative execution from being governed by a specific decision about a branch condition (*i.e.,* should the branch actually have been taken) to a conjunction of an arbitrary collection of constraints on critical ex-

```
0  ...
1    fdivl  %R1, %R2, %R3
2    fadd   %R5, %R6, %R7
3    fcmp   %R1, %R2
4    je     LABEL1
5    jmp    LABEL2
6  LABEL1:
7    movl   $0, %R1
8    movl   $0, %R2
9    movl   %R1, -8(%R30)
10   movl   %R2, -4(%R30)
11   jmp    LABEL3
12 LABEL2:
13 ...
```

Figure 3.1: *Speculative Execution of a Branch.* In this synthetic assembly language, rather than stall the pipeline because of the unresolved result of the floating point divide operation, the processor can choose to issue the floating point add operation on line 2 (out of order execution). If the dependency on `R1` and `R2` between the divide and the compare operations is satisfied, then the compare can execute. Because the result of the compare on line 3 may not be available for the branch instruction in line 4, the processor may speculatively execute (based on branch prediction) the code at `LABEL1` or fall through to the direct jump on line 5. If the branch prediction is incorrect, the speculated instructions are flushed and execution continues from the correct target.

ecution artifacts and other integral data items of the supervised software system. This contribution is a logical complement to superscalar instruction scheduling algorithms like Tomasulo's Algorithm.

In much the same way that a superscalar processor speculatively executes past a branch instruction and discards the mispredicted code path, we propose that the instruction streams of software systems be executed in two logical phases. The first phase executes instructions, optimistically "speculating" that the results of these computations are benign. The second phase makes the effects of the speculated instruction stream visible to the OS and hardware environment. It also potentially enacts a self–healing repair if the effects of the instruction stream have been deemed harmful or corrupted.

The work closest to the notion of policy–constrained speculative execution is the work by Oplinger and Lam; they also propose [OL02] a transactional approach to improve software reliability. Their key idea is to employ thread level speculation (TLS) to execute monitoring code in parallel with the primary computation. The computation "transaction" is rolled back depending on the results of the monitoring code. In contrast, we seek to repair the important data items of the transaction.

3.3 The Clark-Wilson Integrity Model

Microspeculation provides a *mechanism* for executing a self–healing response to a fault or attack, but we still require a model for the *policy* that controls this mechanism. We need, in essence, a model of computation for the policy language and a model that describes the content, composition, and evaluation of the integrity constraints such a policy would be based on. We find almost exactly what we need in the Clark-Wilson Integrity Model [CW87].

Twenty years have passed since the publication of Clark and Wilson's description of a model for information integrity. We assert that the model also applies equally well to execution integrity. The Clark-Wilson Integrity Model (CW) formalizes the notion of information integrity. A policy that uses the model describes the relationships between principals, data items, and operations on those data items. Clark and Wilson developed the model to illustrate that then-current integrity models were better suited to the confidentiality re-

quirements of multi-level security systems. Clark and Wilson use the running example of business accounting principles (*e.g.*, double-entry bookkeeping) throughout their analysis. Their model provides enough formalism to describe how these longtime manual procedures can be implemented in a computerized context.

3.3.1 CW Model Background

The basic data type in CW is a Constrained Data Item (CDI). A CDI has a logical relation constraining its value range. There are two types of procedures that can operate on a CDI. A Transformation Procedure (TP) is a well-formed transaction that transitions the system from one valid state to another. A TP processes sets of CDIs on behalf of a particular authenticated user. An Integrity Validation Procedure (IVP) is used to measure whether a CDI conforms to the integrity specification. In real software systems, TPs are analogous to functions, methods, or routines. In a general sense, IVPs roughly correspond to mechanisms like double-entry bookkeeping in financial systems. For software security, IVPs map to detection mechanisms like stack canaries [CPM+98, Eto00], taint-tracking [CCCR05, NS05], program shepherding [KBA02], address space randomization [BDS03, XKI03] and array bounds checking in languages like Java. In these cases, the CDIs in the system are data constructs like stack-resident return addresses, the instruction pointer, or sensitive system call arguments. CW applies equally well to both these low-level data items and more complex data structures. Of course, not all data in a system is constrained. In these cases, data (*e.g.*, from the user, adversary, or the environment) is represented by an Unconstrained Data Item (UDI). Any TP taking a UDI as input must ensure that it transforms all possible values of a UDI to a CDI. Many system vulnerabilities arise from the operation of an incorrect TP on a UDI, or data derived from a UDI.

The core of the model is a collection of nine rules that deal with enforcement (E) and certification (C) of users, IVPs, TPs, CDIs, and UDIs. For reference, we reproduce the rules below. Some rules are annotated with shorthand names that sum up their purpose.

1. **C1:** All IVPs must properly ensure that all CDIs are in a valid state at the time the IVP is run.

2. **C2:** All TPs must be certified valid. For each TP and each set of CDI that it may manipulate, the security officer must specify a "relation" which defines that execution. A relation is of the form: (TP_i, (CDI_a, CDI_b, CDI_c, ...)), where the list of CDIs defines a particular set of arguments for which the TP has been certified.

3. **E1:** The system must maintain the list of relations specified in rule C2, and must ensure that the only manipulation of any CDI is by a TP, where the TP is operating on the CDI as specified in some relation.

4. **E2:** The system must maintain a list of relations of the form: (USERID, TP_i, (CDI_a, CDI_b, CDI_c,...)), which relates a user, a TP, and the CDIs a TP may reference.

5. **C3:** The list of relations in E2 must be certified to meet the separation of duty requirement.

6. **E3: (Authentication)** The system must authenticate the identity of each user attempting to execute a TP.

7. **C4: (Auditing)** All TPs must be certified to write to an append-only CDI with all information necessary to permit the nature of the operation to be reconstructed.

8. **C5: (Taint)** Any TP that takes a UDI as input must be certified to convert a UDI to a CDI, or reject the UDI, for any possible value of the UDI.

9. **E4: (Separation of Duty)** Only the agent permitted to certify entities may change the list of such entities associated with other entities: specifically, that associated with a TP. An agent that can certify an entity may not have any execute rights with respect to that entity.

The heart of any specific policy expressed in CW terms is the list of the relations specified in rule **E2**. The remainder of the rules simply ensure that the model works as expected. The basis of the model relies on manual certification of TPs and IVPs. Of course, if all such procedures did not contain vulnerabilities simply by virtue of certification, the security community would have little to worry about. Certification is performed by humans on inherently complex constructs and is therefore susceptible to error. Furthermore, the

mechanisms proposed by the model must themselves be protected against tampering. Rule **E4** helps to ensure this, but it still depends on a manual process.

Clark and Wilson note this and suggest minimizing certification rules. They also note that many systems do not separate policy and mechanism: application-specific policy is encoded in TPs (an observation especially true for error-handling code). This observation leads directly to one of our contributions: the separation of repair policy from the mechanism of repair[5]. We split the problem of self–healing into a policy specification problem (something human expertise can be leveraged for) and a policy satisfaction problem (a process that is amenable to automation and at which computers are adept).

The CW model focuses on detecting and preventing unauthorized data modification. Although a TP should move the system from one valid state to the next, it may fail for a number of reasons (incorrect specification, vulnerability, hardware faults, *etc.*). The purpose of an IVP is to detect and record this failure. CW does not address the task of returning the system to a valid state or formalize procedures that *restore* integrity. In contrast, we concern ourselves with ways to recover after an unauthorized modification.

3.3.2 Model Limitations

Clark-Wilson (and our extensions, described more fully in Chapter 5) rely on a number of assumptions. Perhaps the most significant is that each user is only permitted to execute a specific set of programs (TPs). The system should ensure that it is not possible for a user to augment their set of programs to bypass the SoD rules. The phenomena of emergent properties makes this assumption notoriously difficult to guarantee, because future configurations of the system may contain programs that can be combined in unexpected ways. Therefore, system reconfigurations must be certified. This requirement is somewhat unrealistic for current and foreseeable software systems, and complexity often makes the certification problem intractable, even though some utilities such as the Unix `sudo` program seek to provide just this type of enforcement.

[5]This separation can be considered an example of aspect–oriented programming, in which a program is seen as a combination or *weave* of different aspects of a solution: logging, security, calculation, I/O and storage services, parsing, user management, performance management, and resource allocation.

More to the point, this security model relies heavily on the notion of authenticated principals with non-overlapping authorization roles. This requirement, however, has the largest effect on security concerns, since, as Clark and Wilson note, it relies on certification rules. Translating the model to real software is a challenge, especially since the complexity of modern systems threatens to violate many of the underlying security expectations. In our implementations, we make a number of reasonable design choices to illustrate the feasibility of the key concepts, and this thesis refrains from attempting to provide a comprehensive authentication and authorization infrastructure.

Boolean expressions form the basis of one class of repair policy constraints. However, such expressions may only represent a measurement of a high-level property of another data structure or CDI. We term such a relationship a "*state delegate*." For example, a CDI named `num_sorted` may be constrained to be less than the value `10`. Repair likely involves not only modifying the value of the CDI `num_sorted`, but also the adjustment of the position of data items in the underlying CDI. We consider the root cause of this difficulty more fully below and defer the investigation of repair policies for these types of algorithmic relations.

3.4 Constraint Satisfaction

Our hypothesis is that we can use constraints — with a logical organization imposed by the terms of our modified Clark-Wilson Integrity Model — to control a self–healing algorithm in which a pipeline of execution is speculatively executed subject to those constraints. While we believe such a pipeline and system organization provides a solid foundation for building self–healing software, relying on constraints is not a panacea. We must be careful to qualify our hypothesis according to what constraint satisfaction can actually achieve.

Clark-Wilson admits the existence of data items for which no constraining relation exists. If a constraining relationship does not exist, or we cannot succinctly express the constraining relationship in terms of a boolean expression (without arbitrary algorithmic processing or control flow), or we are unaware that we should constrain a particular data item, then we will be unable to protect that data item (except, perhaps, incidentally, as part of the constraint on some seemingly unrelated data item). As noted above, complex, algorithmic

relationships require more intelligent processing than is possible in straightforward boolean constraint satisfaction (*e.g.,* handling state delegates).

We next explore the foundations of constraint satisfaction to ascertain what we can expect to achieve by using the power of automated constraint satisfaction. We close by reviewing our choice of a constraint solver, RealPaver. Our primary source for this Section is the widely used AI text from Russell and Norvig [RN02]. Readers familiar with this material may safely skip the remainder of this Section.

3.4.1 Theory of Constraint Satisfaction

Using machine intelligence to solve problems often involves some type of directed search through a space of solutions. Exhaustive depth–first search provides the basis for mechanisms in this space, but it represents an unsatisfactory solution for all but the most trivial of example or toy problems. Instead, the complexity of most interesting problems and the corresponding size of the solution space requires that search procedures be directed in an intelligent manner, and that an evaluation function must aggressively filter or prune the search space to arrive at a satisfactory solution (either globally or locally optimal) in a reasonable amount of time[6].

Most methods for "intelligent" machine search consist of a rather standard composition of routines. These search methods consist of: (1) a *goal test* that checks whether the state under consideration represents a solution to the problem, (2) an *evaluation* or *heuristic* function for determining the fitness of a particular state in the search space or deriving an estimate of how far that state is from a solution state, and a (3) *successor* function that determines which state or set of states should be evaluated next.

Many such search algorithms abstract or hide the internal details of each state in the solution space. Unfortunately, this design decision implies that the construction of appropriate evaluation functions, heuristic functions, and goal tests often utilize domain–specific

[6]The definition of "reasonable" may vary. For example, generating the upcoming baseball season's schedule may permissibly take a few days or weeks, whereas moving an arbitrary piece of luggage through an airport's baggage handling system must often be done within minutes. Sometimes, of course, such requirements are not met, as suggested by the old saw: "There are two types of luggage: carry-on and lost."

knowledge to access the information contained in the state. In essence, all the routines used to help solve a particular problem are problem-specific and cannot be reused, even for problems that differ only slightly.

Constraint satisfaction problems, on the other hand, employ a state structure in which it is possible to use a standard representation to interact with the subroutines of the search. As a result of each state adhering to a basic, well–defined specification (*i.e.,* variable assignments from a given domain that are subject to boolean constraints), the design and implementation of a goal test, successor function(s), and evaluation function(s) can also be standardized and used interchangeably to solve many different specific constraint satisfaction problems without the need to rewrite or tweak these routines for each new problem or problem instance. The difficulty of a CSP varies with the type of variables involved in the CSP and the type of constraints that govern those variables.

3.4.1.1 Definitions

According to Russell and Norvig [RN02], a constraint satisfaction problem (CSP) involves a set of variables, $X_{1...n}$ and a set of constraints, $C_{1...m}$. Each variable in X has a non-empty domain of values. Each constraint in C imposes limitations on the allowable domain values for some subset of variables in X. The standard representation (alluded to above) of a state in a CSP specifies an assignment of values to a subset of X. If this assignment does not violate any of the constraints in C, then it is called *consistent*. A particular assignment or state need not include every variable; if it does, however, it is called *complete*. An assignment that is both complete and consistent is a *solution* to the CSP.

3.4.1.2 The Nature of Variables and Constraints

Perhaps the simplest type of CSP involves discrete variables that have a finite domain: variables that can take only a few well–defined countable values. An example of a finite-domain CSP involving discrete variables may be selecting a particular shirt and tie combination from the closet.

$$Variables = \{X_1 = shirt,\ X_2 = tie\}$$

$Values = \{blue,\ black,\ polkadot,\ green,\ red,\ white\}$

The variables in this case are the shirt and tie, and the values include some finite set of available colors. Note that we have yet to impose constraints on the variables. With only a set of variables and their possible values, no problem yet exists. A credible constraint may restrict the use of polkadot in any solution, as we will see below.

Although Russell and Norvig caution that "we cannot expect to solve finite-domain CSPs in less than exponential time" [RN02], this advice does not represent cause for worry. For example, 3SAT is a special case finite-domain CSP (variables are booleans with a finite domain that includes the values `true` or `false`), and we know 3SAT to be NP-complete. Even though such CSPs take exponential time to solve in the worst case, *they remain decidable* — even if the decision is reached through an exhaustive, exponential time search. The size of the problems that we consider, however, have only a few tens of variables at most, so we do not anticipate lengthy runtimes of the constraint solver. Finally, because of the standard way of representing states, and since CSPs have the property of *commutativity* (that is, it does not matter what the order of assignment of variables is), it is possible to use constraint programming to solve problems that are orders of magnitude larger than those solvable with basic search algorithms, as "hard" CSPs are actually quite rare [RN02].

CSPs involving discrete variables can also include discrete variables with an unbounded or infinite domain. Integers are an example of this type of variable. For these types of CSPs, we need to use some type of constraint language to represent problems, because we can no longer simply enumerate all possible combinations of domain values. In some cases, we can bound the domain of an infinite domain CSP (the bound simply acts as another constraint). Our repair policy language, Ripple, and the constraint solver we use, RealPaver (discussed below), are two different examples of such a constraint language. Finally, variables may also be drawn from continuous domains (*e.g.,* members of $\Re$).

The structure of constraints has some implications for solving CSPs. Unary constraints (similar to a unary operator in a programming language) only involve one variable and can be reduced by removing any domain value that violates the constraint. Binary constraints, as the name implies, involve a relation between two variables. Higher-order constraints (*i.e.,*

those involving three or more variables) can be reduced to a series of binary constraints if enough temporary or auxiliary variables are introduced.

In our shirt and tie example, we may have unary constraints that remove the values *polkadot* and *blue* from the domain of allowable values.

$$Constraints = \{C_1 = !polkadot,\ C_2 = !blue,\ \ldots\}$$

After applying these constraints to reduce the domain values

$$Values = \{black,\ green,\ red,\ white\}$$

we can remove them from the constraint set. In our shirt and tie example, the reduced constraint set may now look something like this:

$$Constraints = \{C_1 = (X_1{=}black \wedge X_2{\neq}white),\ C_2 = (X_1{=}green \wedge X_2{\neq}red)\}$$

Solutions to the CSP include the assignments $(X_1/black,\ X_2/red)$ and $(X_1/red,\ X_2/white)$.

Although linear constraints on integer variables are a well–studied problem, Russell and Norvig note that "no algorithm exists for solving general nonlinear constraints on integer variables." Non-linear constraints approach the realm of arbitrary algorithmic operation on variables, and we expect (inline with fundamental computability results such as Rice's theorem) that such computation is beyond the facility of an automated process to decide. As a small example, Figure 3.2 shows that a constraint involving an arbitrary function call might be impossible to satisfy, as the function in question may never return. In this thesis, we only consider constraints involving data items directly rather than constraints that involve arbitrary control flow.

3.4.2 RealPaver

Rather than writing a constraint solver from scratch, we adopted a third party constraint solver called RealPaver [Gra04]. Our self–healing system uses our repair policy language

```
...
int x = LAUNCH_OK;
Object data;
...
assert(x == launchData(data));
...
}
```

Figure 3.2: *Constraint with Arbitrary Control Flow.* Constraints that involve arbitrary control flow may not terminate; it is therefore impossible to decide CSPs involving these expressions. Here, we cannot guarantee that `launchData()` ever returns. Consequently, an automated process for satisfying this constraint cannot guarantee that it will halt.

and its runtime environment to communicate with and obtain solutions from RealPaver. RealPaver defines a modeling language for solving systems of numerical constraints. An example of a short RealPaver program is shown in Figure 3.3.

```
Constants
  authfailurecode = 0  ; /* semantically valid EV value */

Variables
  int rvalue in [-255, 256] ; /* return value is within this range */

Constraints
  rvalue = authfailurecode;
```

Figure 3.3: *Example RealPaver Program.* This RealPaver program expresses one constraint; that the variable `rvalue` must be equal to `authfailurecode`. More complex, multi–clause constraints are possible, such as sets of linear and non–linear equations.

Each model in the language defines a CSP in terms of a set of real-valued variables X, a set of domains, and a set C of constraints over the set of variables. RealPaver is *reliable* in that it will find a solution if one exists. Given a CSP expressed in RealPaver's modeling language, it computes a union of bounding boxes that contains all the solutions of the CSP. RealPaver can handle constraints involving arbitrary mathematical expressions.

3.5 Error Virtualization

The key assumption underlying error virtualization is that a mapping can be created between the set of errors that *could* occur during a program's execution and the limited set of errors that the program code explicitly handles. Figure 3.4 displays an example function where a buffer overflow can be caught and mapped to a sibling error that detects a NULL input buffer. By virtualizing errors, an application can continue execution through a fault or exploited vulnerability by nullifying its effects and using a manufactured return value for the function where the fault occurred.

```
0   int bar(char* buf)
1   {
2           char rbuf[10];
3           int i=0;
4           if(NULL==buf)
5                   return -1;
6           while(i<strlen(buf))
7           {
8                   rbuf[i++]=*buf++;
9           }
10          return 0;
11  }
```

Figure 3.4: *Error Virtualization.* We can map unanticipated errors, like an exploit of the buffer overflow vulnerability in line 8, to anticipated error conditions explicitly handled by the existing program code (like the error condition return in line 5). While this example is shown in C code for clarity, the operation of the tools and systems discussed in this thesis happens at the machine instruction level.

In previous versions of our system [SLBK05], these return values were determined by source code analysis on the return type of the offending function. Basic error virtualization seems to work best with server applications — applications that typically have a request processing loop that can presumably tolerate minor errors in a particular iteration. In

contrast, this thesis provides a practical solution that improves the semantic correctness for client applications (*e.g.,* email, messaging, browsing) and servers.

Both error virtualization [SLBK05] and failure oblivious computing [RCD+04a] prevent exploits from succeeding by masking failures. According to the results from its experimental evaluation, however, basic error virtualization fails from 10% to 12% of the time [SLBK05] by causing a crash or hard termination of the monitored process. Furthermore, both approaches have the potential for semantically incorrect continuation of execution. These shortcomings are devastating for applications that perform precise calculations or, for example, provide authentication and authorization services. This limitation is especially relevant for financial and scientific applications, where a function's return value may be incorporated into the mainline calculation.

Furthermore, error virtualization in this form requires access to the source code of the application to determine both appropriate error virtualization values and proper placement of the calls to the supervision environment. A better solution would operate on unmodified binaries and provide a way to customize error virtualization values during runtime. The work that this thesis presents accomplishes these tasks.

3.5.1 Revisiting the EV Hypothesis

Early results on the use of error virtualization were encouraging. For three popular open–source server applications (Apache, Bind, sshd), Sidiroglou *et al.* [SLBK05] report that roughly 90% of the *leaf* functions (*i.e.*, functions without children) in these applications are amenable to error virtualization. In these previous experiments, the authors define success as the application not crashing after a function undergoes error virtualization. While these initial results seem positive, from another perspective, they may be devastating: the technique fails about 12% of the time, and "not crashing" provides no confidence as to the semantically correct continuation of execution. Furthermore, applying error virtualization to leaf functions may set up the technique for optimal results, as slicing off a leaf function does not remove an entire subtree of execution. The previous experimental setup implies that the success rate is likely the maximum, given a set of applications (network servers) that can tolerate errors in one request/response cycle.

Error virtualization assumes, in the best case, that most portions of an application check the return values of invoked routines. The prevalence and extent of such error handling code is questionable. Furthermore, in any given software system, the error virtualization hypothesis is not applicable to a few core routines because the absence of these routines from the execution stream neutralizes the core functionality of the system. In short, without these routines executing properly, no amount of error handling can gracefully recover the application and provide full service to the user of the system. Essentially, repeated use of widespread error virtualization *crystallizes* — or makes brittle — the execution of a program by eliminating functionality and reducing the program to a series of simple routine calls and projected return values. In this situation (admittedly, the logical extreme), the application has, in effect, been replaced by a function that generates a sequence of numbers.

3.5.2 Background: Memoization

This type of program behavior is similar to the concept of *memoization*. Memoization is the process of caching a map of the results of a function to the argument values of the function. Thus, each instance of the function is translated to a simple lookup. Memoization is an optimization technique, and its correctness depends on each function being idempotent; that is, having no side effects. Memoization as a term was introduced by Donald Michie in 1968[7]. In essence, a particular instance of a function call (as defined by the values of its parameters) is stored in a lookup table so that future invocations of that function with the same arguments reduce to a lookup of the return value in the table. This mechanism has clear links to the concept of dynamic programming and other forms of caching. The obvious benefit of memoization is a speedup in execution, but this speed benefit comes at the cost of space to store the results of previous function invocations. For error virtualization, this cost is augmented by the penalty of incorrect "memoization": semantically incorrect continuation of execution.

A function can only be memoized if it is referentially transparent[8]: only if calling the function will have the same effect as replacing that function call with its return value.

[7] `http://www.aiai.ed.ac.uk/~dm/dm.html`

[8] Functional languages are built on routines that have no side effects, unlike procedural languages like C.

Clearly, many modern real-world applications written in imperative programming languages like C, C++, C#, and Java have a number of functions that could be memoized. Just as clearly, this assumption does not hold for all functions in these types of software applications, as many useful programs perform some I/O or modify global state from within individual functions[9]. In the case of errors, simply returning an error value may not be enough, as some global state cleanup may be needed to make the function "referentially transparent." Providing this kind of cleanup is one feature of repair policy.

3.5.3 Evaluating the Range of Error Virtualization

Faults and vulnerabilities often lie along code paths that are infrequently exercised. As such, the developer may have paid little attention to its specification and testing, and the administrator or user may not stumble upon its failure mode simply because they are unaware of its existence or it lies outside of their normal use profile. Such code paths include functionality that is, by definition, largely unimportant to the common use cases of an application. Such routines are therefore prime candidates for being treated as abortable transactions. While the majority of individual leaf functions, once subjected to error virtualization, may indeed behave as cleanly aborted transactions, there are, as we mention above, at least a few critical routines which will not. It is an open question whether or not these core routines are likely to contain a greater or lesser number of vulnerabilities than the rest of the code base. Consequently, it is difficult to say whether they would frequently be in need of error virtualization even if they are the routines that can tolerate it (from the viewpoint of actually accomplishing the purpose of the system) the least.

We hypothesize that as more routines are subjected to error virtualization, the greater the chance of the system failing because one of these "critical" routines has been encountered. On the other hand, it is likely that only a relatively small number of all total functions need ever experience error virtualization, and thus the pathological case of all functions undergoing EV should not be seen in practice. Furthermore, since core or critical routines are

[9] As an obvious example, the program statement `printf(``Hello, world.'');` is not referentially transparent, since replacing this call with its return value (the number of characters printed) does not yield the same program ("Hello, world" is not printed).

regularly exercised, we *may* have more confidence in their level of robustness, and therefore be able to more easily apply error virtualization. In any event, if a program is replete with exploitable errors, then continuing execution is hopelessly optimistic; instead, the system should be halted and taken offline for a thorough repair or complete rewrite.

Identifying the routines that cannot be cleanly subjected to vanilla error virtualization can help drive the generation of repair policy. Once we know these routines do not behave well under vanilla error virtualization, they are candidates for a developer, repair policy guru, administrator, or user to focus on developing a repair procedure. The purpose of the experiments in this Section is to illustrate a methodology for identifying candidates for those critical functions.

In the following experiments, we need to have a well–defined notion of program "correctness." The goal of error virtualization is to keep the majority of the program's core functionality undisturbed. For example, causing a logging routine in a Web server to employ error virtualization should not prevent the delivery of the requested page to the client. There is, however, a scale of possible behaviors that can result from the process of error virtualization. This scale of behavior ranges from coarsely correct toward a very fine-grained notion of correct program execution: (a) not crashing the application, (b) not raising any externally noticeable events that were not previously visible, (c) execution behavior becomes correct with respect to the programmer's implementation, (d) execution behavior becomes correct with respect to the programmer's intent, (e) correct with respect to the programmer's understanding, and (f) absolute correctness.

It is difficult to empirically measure the last few correctness levels. In the following experiments, we use the correctness scale listed in Table 3.1. Note that the last case mentioned above (case f) is beyond optimal: a programmer's mistake may be fixed. We are unlikely to see this last type of event in practice (or even a contrived setting) because error virtualization does not generate new code[10]; it simply redirects execution along existing code paths.

[10]In some very rare cases, it may in fact activate code that has been dormant, thus simulating the addition of "new" code to the execution stream.

Table 3.1: *Scale of Correctness for Error Virtualization.* Error virtualization is not a sound technique. It has the potential to disrupt the application semantics even as it attempts to recover those semantics in the face of unanticipated failures. We list a scale of overall correctness for programs that have been subjected to error virtualization. This discrete scale is used as the vertical axis for the graphs in Section 3.5.5. Note that the relative magnitude of each step is difficult to determine. In the list below, "behavior" consists of output *and* simple behavior features like instruction counts, function invocation counts, and memory write counts.

0	application crashes
1	application does not crash (may hang or loop)
2	application finishes execution (no hang)
3	application finishes, correct *program* return value
4	application finishes, and behavior matches $< 25\%$
5	application finishes, and behavior matches $< 50\%$
6	application finishes, and behavior matches $< 75\%$
7	application finishes, and behavior matches 100%
8	optimally correct (fix underlying fault)

3.5.4 Experimental Setup

The purpose of these experiments is to evaluate the efficacy of error virtualization in light of previously reported experimental results. We want to see whether error virtualization is applicable to types of applications other than network servers and whether it can provide stronger guarantees than those initially envisioned for those types of programs. Consequently, we evaluate the basic error virtualization hypothesis for a variety of command–line applications listed in Table 3.2. If these stronger guarantees cannot be achieved reliably, then this result motivates the need for a new approach.

We collect valid outputs for a run of each of these programs. Note that it is inherently difficult to automatically compare outputs from different runs of programs, forcing us to

Table 3.2: *List of Microbenchmarks for Error Virtualization.* A list of programs that we execute subject to error virtualization.

Benchmark #	Benchmark Name	# of Functions
1	`arch`	51
2	`true`	200
3	`hostname`	204
4	`sort /etc/passwd`	768
5	`md5sum vmlinuz`	1491
6	`echo ‘‘hello, world.’’`	204

limit the scope of our evaluation[11]. For example, `date`, `netstat`, and `ps` will display different valid output between runs. Even something as simple as `pwd` will display different behavior if the test phase is not executed in the same directory as the valid output collection phase[12]. In cases like these, we had to manually examine the output for indications of faulty execution post-error virtualization.

3.5.5 Individual EV

Overall, individual functions may respond well to error virtualization. We next present some representative graphs of the results of applying error virtualization to individual functions in some of our microbenchmarks. We force an error virtualization response by writing the value `0xffffff` into the `%eax` register and rolling back all memory changes made during the target routine. Doing so effectively slices off that function from execution. Sources of error include undoing memory writes that affect I/O and overwriting an `%eax` value that is not checked (*i.e.*, return from a `void` function). We could improve the fidelity of the test

[11] A comprehensive examination of the error virtualization hypothesis is outside the scope of this thesis. Running these experiments, even in an automated fashion, is time-consuming ($O(N^2)$, where N is the number of slices in an application) because we run the application N times for each “failed” slice. We only run these experiments to shed some light on the power of the technique.

[12] Of course, setting up a virtualized environment that can rewind to a clean default state after every test can help, but the amount of effort to do so is unwarranted for the purposes of this Section.

if we added a check to our system that detects a read operation of the `%eax` register at the fall-through instruction for a returning routine.

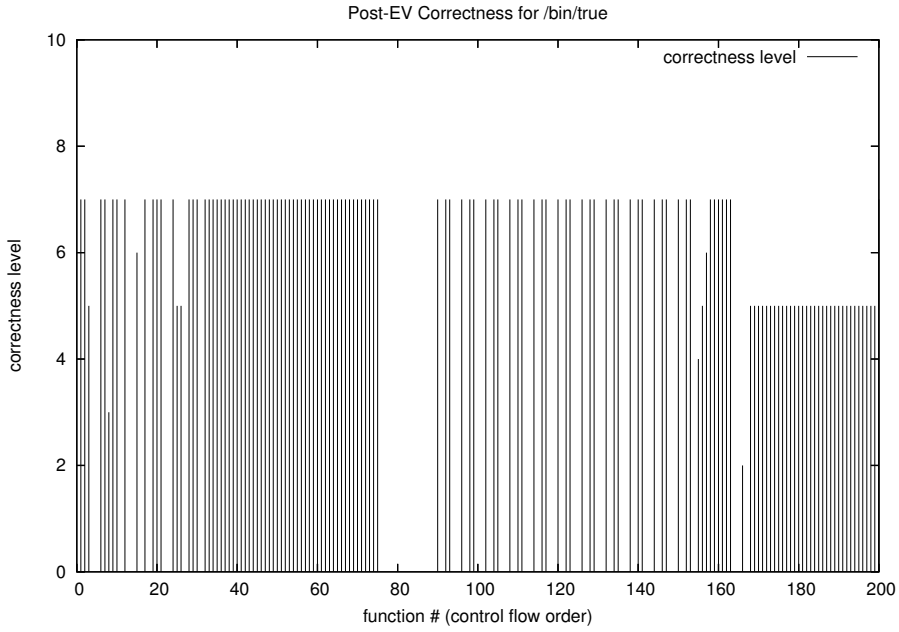


Figure 3.5: *Correctness for Individual Functions of* `true`. Gaps indicate post-EV crashes.

In Figure 3.5, the horizontal axis represents the slice (*i.e.*, function) number, in control flow order, of `true`. The vertical axis plots the success level with values drawn from Table 3.1. For this application, most functions do not tolerate vanilla error virtualization. Table 3.3 contains the distribution of slices and their correctness results. This program contains 200 executed function instances. Half of all instances (99 of 200) are amenable to vanilla error virtualization. Of the other function instances, 60 crash the application after undergoing error virtualization, and the remaining 41 perturb the execution to produce results somewhat less than completely correct. Of the 99 "correct" instances of error virtualization, since `true` is a short-running application, we were unable to determine if subtle errors occur which might negatively affect the execution of a long-running program. We are also unsure if the remaining 41 also introduce subtle errors, but given that they introduce short-term variation, the introduction of errors in the long run seems very likely. Although one may argue that crashing is a good result (in that the symptoms of a failure become

fairly noticeable), the aim of self–healing is to avoid both crashes as well as semantically incorrect follow-on execution.

Table 3.3: *Distribution of Correctness Results for* `true`

Number of slices	**Percentage (n=200)**	**result**
99	49.5%	(7) successfully experience EV
60	30%	(0) crash
36	18%	(5) about half the output differs
2	1%	(6) most output matches
1	0.5%	(4) most output differs
1	0.5%	(3) execution corrupted
1	0.5%	(2) results differ greatly

In similar fashion, we analyzed the behavior of the `hostname` utility. This utility exhibits roughly the same behavior as the instance of `true` that undergoes error virtualization (see Figure 3.6). Like `true`, only about half of the functions (102 of 204) can be said to successfully tolerate error virtualization, but these 102 function instances did not display equivalent behavior to an unperturbed `hostname` instance. Instead, only 3% (6 of 204) of the functions actually error virtualize completely correctly. About 29% of the functions crash the program once subjected to error virtualization, and another 16% exhibit noticeably different execution traces. See Table 3.4 for more detail.

Table 3.4: *Distribution of Correctness Results for* `hostname`

Number of slices	**Percentage (n=204)**	**result**
102	50%	(6) most output matches
59	28.9%	(0) crash
34	16.7%	(5) about half the output differs
6	2.9%	(7) successfully experience EV
2	0.9%	(4) most output differs
1	0.5%	(2) results differ greatly

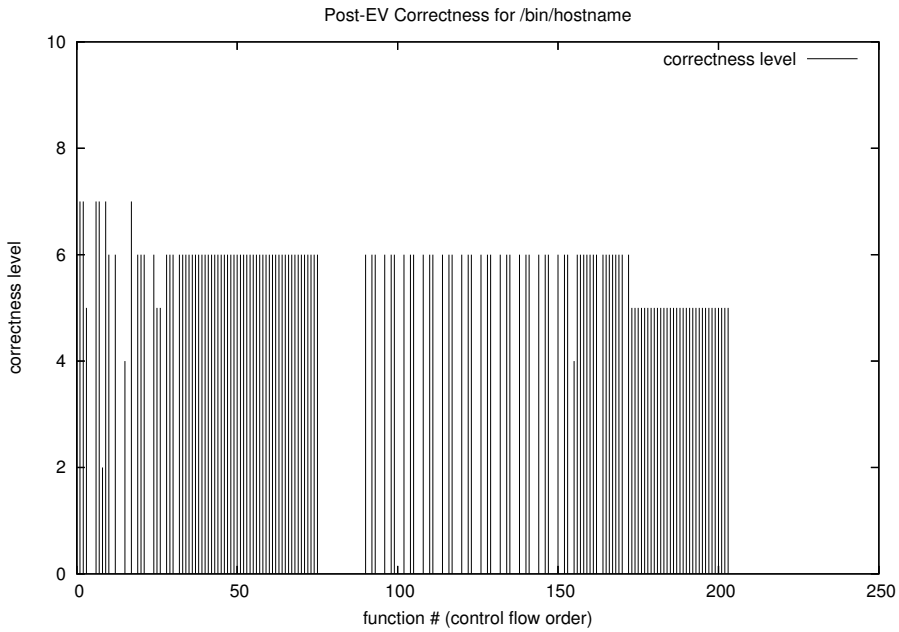


Figure 3.6: *Correctness for Individual Functions of* `hostname`. Gaps indicate post-EV crashes.

3.5.6 Summary

It is not the purpose of this thesis to fully evaluate the error virtualization hypothesis. We perform the limited experiments of this section as a way to examine the efficacy of the basic technique and provide motivation for our repair policy work.

Error virtualization of dynamic traces of a variety of real software shows a wide range of success, although runs that approached the correctness of a pristine run were not overwhelmingly common. We observed a number of errors arising from the effects of error virtualization. In most of these cases, the application crashes due to a segmentation violation (SIGSEGV). In a few cases, we observe a SIGABRT due to a double free or memory reallocation error (invalid size). In at least one case, the `sort` program entered an apparently infinite loop. We manually terminated the run (function 484) after the test output soaked up the remaining disk space on the test machine.

Previous experiments showed promise but were limited in scope (they cut off leaf functions for server programs). This experimental setup allows (but does not guarantee) error

Table 3.5: *Distribution of Correctness Results for* `arch`. We measure success by seeing if the program outputs `i686`, the architecture of our test machine.

Number of slices	**Percentage (n=51)**	**result**
43	84.3%	(7) successfully experience EV
8	15.6%	(0) crash

Table 3.6: *Distribution of Correctness Results for* `sort` *of a user database file.*

Number of slices	**Percentage (n=768)**	**result**
319	41.5%	(7) successfully experience EV
348	45.3%	(0) crash
100	13.1%	(4) most output differs
1	<1%	(1) application loops

virtualization to display best case behavior because whole branches of execution are not lost. The ASSURE approach [SLKN07] (identifying rescue points through an offline dynamic training process) is a different way to help improve the efficacy of error virtualization.

Further experimentation is needed to detect the effective radius of error virtualization on a per-application basis. As we discuss above, these experiments should start crystallizing the whole execution, beginning at various points in program execution. These experiments can produce graphs showing the overall decrease in functionality as more functions are subjected to error virtualization. The horizontal axis would contain each slice of a dynamic trace, and the vertical axis would plot the average distance that error virtualization can succeed before the application fails. Future experiments should aim to discover and characterize the effective "radius" of error virtualization: that is, how much error virtualization software systems can suffer before becoming unable to supply their core functionality.

Table 3.7: *Distribution of Correctness Results for* `echo`. Even a utility as simple as this crashes more than a quarter of the time after error virtualization.

Number of slices	**Percentage (n=204)**	**result**
145	71.1%	(7) successfully experience EV
59	28.9%	(0) crash

Table 3.8: *Distribution of Correctness Results for* `md5sum` *on a Linux kernel image.* About 2% of the routines, after undergoing error virtualization, produce an incorrect MD5 value for the file. Almost a third crash the utility. Of the roughly 1011 which we rate as surviving, we do so because they produce a valid MD5 value; we do not evaluate whether the execution matches a pristine run exactly (in terms of instructions executed and memory writes).

Number of slices	**Percentage (n=1491)**	**result**
1011	67.8%	(7) successfully experience EV
443	29.7%	(0) crash
37	2.4%	(2) results differ greatly

Chapter 4

Microspeculation Runtime Environments

Microspeculation (*i.e.,* selective, policy–constrained speculative execution) is the fundamental mechanism we use to supervise the execution of applications we wish to protect (we provide background information on microspeculation in Section 3.2). Our goal is to provide automated defense by speculatively executing slices of a software system that we either expect to be or suspect of being "weak" (defined as containing a fault or vulnerability). Each slice effectively represents a transaction. We therefore needed to construct a runtime environment that supported the ability to execute arbitrary slices of a software application in a speculative fashion (undoing the effects of the slice or transaction as needed). This runtime environment would, in essence, be easy to turn on and off, switching seamlessly between supervised execution and normal execution.

This chapter describes two microspeculation systems we have developed: the first and second versions of STEM (Selective Transactional EMulation). Both runtime environments support the microspeculation of slices of a software application in a configurable fashion. We first developed a library, STEMv1, that applications could link against to get "supervised computation" services. The second (STEMv2) uses dynamic binary rewriting to insert supervision code into program slices (*i.e.,* transactions). Both instances add a policy-driven layer of indirection to an application's execution.

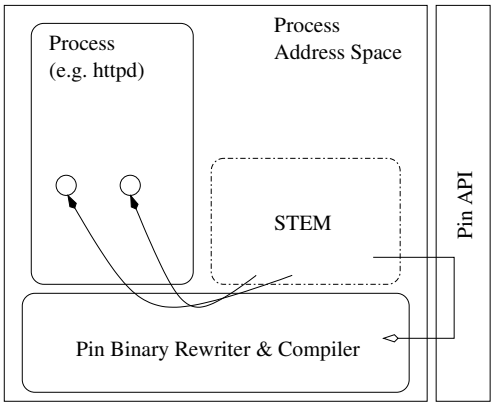


Figure 4.1: *STEMv2 Architecture*. STEM is a *Pin tool*: it uses Pin's API to inform the binary rewriter when to insert it into execution. Pin, STEM, and the original application execute in the same process address space.

4.1 Slice Selection Strategies

Making an *a priori* determination of which slices or portions of a program may be "weak" and thus require monitoring by STEM is an interesting challenge. After all, if we know that a particular piece of code contains a vulnerability, why not simply fix the vulnerability and thereby eliminate both the weakness and the need for supervision?

First, although we cannot reliably identify all vulnerabilities prior to deployment, we may be able to hypothesize that certain code sections are prone to error or susceptible to attack. In short, we may suspect these code sections or control flow paths of containing vulnerabilities, but cannot perform the exhaustive testing necessary to confirm this fact, especially in very large software systems.

Second, for many vulnerabilities, we may not immediately know how to fix the underlying problem (at the very least, humans are unable to do so in the time it takes for an attack to exploit the vulnerability), or the fix may take some time to test. During this period of time, we may still need to have the software system execute and provide service. Therefore, we would like to temporarily protect the vulnerable code slice with some short-term repair.

There are three main ways we can identify potentially vulnerable or faulty code slices. These methods provide the rough boundaries of the slice to be supervised. One convenient

boundary is the encapsulating function, although we can use other discrete boundaries.

First, we can employ standard static analysis tools to detect common programming errors, code–based security violations, and fault conditions. Second, we can construct a shadow, replica, or victim copy of the application. We can observe this heavily–instrumented honeypot for occurrences of a fault or failure[1] and extract the location of the fault from the instrumentation, retroactively placing calls to our supervision environment around this "weak" slice [SLBK05]. This technique can help evolve an application toward a state of protection against that particular fault or vulnerability.

Third, we can rely on statistical measures of vulnerability or other statistical program analysis techniques. The CoSAK (COde Security Analysis Kit)[2] study found that most vulnerabilities in a medium–sized (about 30) set of popular open source software occur within six function calls of an input system call. If one considers a layer or two of internal application processing and the existing (but seldom thought of from an application developer's standpoint) multiple layers within C library functions, this number makes sense. Many vulnerabilities, especially binary code injection vulnerabilities, are exercised by processing external input contained in network packets or file data, so one would expect that vulnerabilities lay relatively close to the input boundary of the software system. Finally, a recent study [NZZ07] found that vulnerabilities correlated well with the revision history and set of files included in a source file containing a vulnerability. In effect, these characteristics predict where the same developer committed the same mistakes or copied the same piece of code to another location (thus requiring the support code exported in the header or included files).

Given any of these strategies, we can selectively protect an application. Of course, if the cost of supervision is decreased by a very efficient or otherwise optimized implementation of the supervision framework, then complete supervision (*i.e.,* 100% coverage of all execution paths) may be practical. We show that even a fairly unoptimized implementation of STEM (STEMv2) using binary rewriting imposes a performance penalty that is not unreasonable

[1] Attackers are motivated to identify and exercise vulnerabilities. As they discover and exploit them, we can notice these events and reactively protect software with a short-term repair.

[2] `http://serg.cs.drexel.edu/cosak/index.shtml`

for supervising a variety of real applications. Nevertheless, security functionality is usually a tax on the user's experience; from a pragmatic point of view, identifying code that does not need to be supervised can only improve the adoption of various supervision mechanisms by reducing the perceived cost of these techniques. Selecting a slice to monitor is more cumbersome with STEMv1; to wit, the source must be edited and the application recompiled. STEMv2 includes a flexible policy to tune the coverage dynamically during runtime. We expand on these points below.

4.2 Microspeculation Using Emulation: STEMv1

Needing to execute arbitrary slices of an application, we examined existing binary supervision environments like Valgrind[3], but none were explicitly built to repeatedly detach and reattach to a software application. Many (*e.g.,* Bochs, VMWare) were too heavyweight in that they were designed to provide a whole–system emulation environment meant to run entire guest operating systems, not slices of a single process. More recent tools like QEMU [Bel05] (an efficient, whole system emulator) and Pin [LCM+05] (a binary rewriting and code instrumentation framework) were not yet available when we set out to construct the first version of STEM.

We designed this first version of STEM as an application–level library that supplies a virtual CPU and an API for invoking the services of (*i.e.,* switching to and from) that virtual CPU. Figure 4.2 provides an example of how a program would use this API.

4.2.1 Instruction Set Randomization

STEMv1 primarily employs a form of artificial diversity, Instruction Set Randomization (ISR), as a sensor to detect binary code injection attacks. ISR is a concept introduced by Barrantes *et al.* [BAF+03] and Kc *et al.* [KKP03]. These approaches can be combined with address-space obfuscation [BDS03] to prevent "jump into libc" attacks. ISR itself is the process of creating a unique execution environment to effectively negate the success of code–injection attacks. This unique environment is created by performing some reversible

[3] `http://www.valgrind.org/`

```
...
emuOptions_t stem_options;
stem_options.hostname = "localhost";
stem_options.port = 3780;
stem_options.frame_radius = 36;
stem_options.config_filename = "stem-cat.conf";
emulate_init();
emulate_begin(&stem_options, 0, -1);
do{
   len = read(fin, buffer, buffer_length);
   write(1, buffer, len);
   total_chars+=len;
}while(len);
emulate_end();
emulate_term();
return total_chars;
```

Figure 4.2: *STEMv1 API.* A snippet of a program that behaves like the Unix `cat` utility. The calls to the STEMv1 API surround the main functionality of the program. STEMv1's API consists of a set of procedure calls and a configuration options type. More details on the purpose of the `frame_radius`, `port`, and `hostname` configuration options are available in Chapter 6. The `config_filename` option provides the filename containing the program slice to ISR key mappings for the program.

transformation on the instruction set; the transformation is driven by a random key for each executable. The binary is then decoded during runtime with the appropriate key.

Since an attacker crafts an exploit to match some expected execution environment (*e.g.*, *x86* machine instructions) and the attacker cannot easily reproduce the transformation for his exploit code, the injected exploit code will most likely be invalid for the specialized execution environment. The mismatch between the language of the exploit code and the language of the execution environment causes the attack to fail. Without knowledge of the key, otherwise valid (from the attacker's point of view) machine instructions resolve to invalid opcodes or eventually crash the program by accessing illegal memory addresses.

4.2.2 Selective Instruction Set Randomization

Randomizing an instruction set requires that the execution environment possess the ability to derandomize or decode the instruction stream during runtime. For machine code, this requirement means that either the processor hardware must contain the decoding logic (current commodity hardware does not) or that the processor be emulated by software capable of derandomization. Emulating the entire execution of a program or operating system is quite expensive in terms of performance: each machine instruction is executed by software. We have measured whole–program emulation to impose up to a 3000% slowdown. STEMv1, however, performs *selective* ISR to reduce this performance penalty by only supervising portions of code we suspect of being weak or containing vulnerabilities.

If we make the assumption that the entire program — that is, every single instruction in the program — is not part of any given vulnerability, and that the amount of good code far outweighs the amount of vulnerable code, then whole–program emulation and supervision is not necessary. Others have also reached this conclusion, and Newsome *et al.* [NBS06] retroactively harden a particular path slice in a program binary to reduce the cost of tainted dataflow analysis: a program supervision technique that requires complete coverage to be effective (missing a taint propagation step can potentially lead to a false negative).

Because it may not be necessary to supervise the entire program, one way of reducing the performance penalty of ISR is to selectively supervise portions or slices of a program according to the slice selection strategies we discussed in Section 4.1. We designed STEMv1 as a user–level library that can selectively supervise portions of an application; all a developer needs to do is include the calls to STEMv1 in the application's source code and link against the library. Execution switches freely between supervised execution on STEM's virtual CPU and native execution on the hardware CPU. This practical, selective form of ISR is one of the technical contributions of this thesis.

STEMv1 effectively supports two different instruction sets at the same time. Various processors support the ability to emulate or execute other instruction sets. These abilities could conceivably be leveraged to provide hardware support for ISR. For example, the

```
0x0804825d <foobar+117>:  call   0x80491c0 <emulate_begin>
0x08048262 <foobar+122>:  add    0x10,%esp
0x08048265 <foobar+125>:  mov    0x8(%ebp),%eax
0x08048268 <foobar+128>:  mov    0x8(%ebp),%edx
0x0804826b <foobar+131>:  mov    (%edx),%edx
0x0804826d <foobar+133>:  add    0xa,%edx
0x08048270 <foobar+136>:  mov    %edx,(%eax)
0x08048272 <foobar+138>:  call   0x8049990 <emulate_end>
```

```
0x0804825d <foobar+117>:  call   0x80491c0 <emulate_begin>
0x08048262 <foobar+122>:  and    0x63,%al
0x08048264 <foobar+124>:  mov    0x2c,%bh
0x08048266 <foobar+126>:  loop   0x8048217 <foobar+47>
0x08048268 <foobar+128>:  sub    0xf2,%al
0x0804826a <foobar+130>:  scas   %es:(%edi),%eax
0x0804826b <foobar+131>:  sub    0xb5,%al
0x0804826d <foobar+133>:  and    0x65,%al
0x0804826f <foobar+135>:  lods   %ds:(%esi),%eax
0x08048270 <foobar+136>:  cs
```

Figure 4.3: *Program Slice Randomization.* Randomizing the first code slice by XOR'ing each byte with the value `0xA7` produces the second sequence of instructions. Randomization is accomplished with our `objrand` tool in a static fashion by randomizing from a start address (in all cases, the address immediately *after* the CALL to `emulate_begin`) to an end address (likewise, the address immediately before the CALL to `emulate_end`).

Transmeta Crusoe chip[4] employs a software layer for interpreting code into its native instruction format. The PowerPC chip employs "Mixed-Mode" execution[5] for supporting the Motorola 68k instruction set. Likewise, the ARM chip can switch freely (from one instruction to the next without any context switch) between executing its "native" instruction set and executing the similar Thumb instruction set. A processor that supports ISR could use a similar capability to switch between executing regular machine instructions and randomized machine instructions. Having hardware support for ISR would obviate the need for (along with the performance impact of) software ISR.

4.2.3 The `objrand` Tool

The ability to selectively derandomize portions or slices of an application's execution is predicated on the ability to *randomize* those portions prior to execution. Running a randomized binary will not succeed; once the hardware encounters a randomized slice, the process will most likely crash due to an illegal opcode or illegal memory dereference (the same conditions that injected binary malcode would produce). The crash happens because the hardware does not know how to derandomize the instruction stream of the randomized slice. Figure 4.3 illustrates how randomizing a program slice, consisting of a valid instruction sequence, produces a completely different instruction stream.

Selective randomization of a program binary requires three things. We must first identify a set of program slices according to the slice selection strategies in Section 4.1. Second, the developer must insert calls to STEMv1 into the source code at the appropriate locations: the slice boundaries. Finally, the developer must compile the application and then extract the addresses of the beginning and end of the slice. One method of doing so is to view a disassembly dump of the program binary in a debugger like *gdb* or *ddd*, although a tool could easily be written to automate this procedure and the generation of the configuration file. This tool is a matter of engineering that we defer to future work. The developer then enters this information into a configuration file (an example of such a configuration file is shown in Figure 4.4).

[4] `http://www.transmeta.com/tech/microip.html`

[5] `http://developer.apple.com/documentation/mac/runtimehtml/RTArch-75.html`

```
# Object Randomization Configuration File for 'httpd' (protocol.c)

function_name = exploitMe
start_address = 0x0809fb47
end_address   = 0x0809fb5b
isr_key       = 0x1
tagnote       = apache test, try out approaches here

function_name = exploitMe_embedded
start_address = 0x0809f68b
end_address   = 0x0809f6a1
isr_key       = 0x2
tagnote       = call mlo_malcode directly @ 0x81d4040

# raddrP is global var @0x8208874
function_name = exploitMe_stackAssist
start_address = 0x0809f7ff
end_address   = 0x0809f862
isr_key       = 0x5
tagnote       = overwrite stack with URI data, manually overwrite raddress

function_name = exploitMe_stackCorrupt
start_address = 0x0809f9be
end_address   = 0x0809f9f4
isr_key       = 0xA
tagnote       = write '0xDEADBEEF' all over the stack
```

Figure 4.4: *An example* ***objrand*** *configuration file.* This particular file is for a randomized version of Apache's `httpd` to be run with some FLIPS experiments (see Chapter 6 and Chapter 8).

The configuration file consists of a series of "program slice" → "ISR key" tuples (we refer to these as SliceKeyTuple mappings). In accordance with standard Unix configuration file practice, lines beginning with a "`#`" character are ignored by the parser. Each SliceKeyTuple consists of five lines of key/value pairs. The randomization is driven by the values of the `start_address` and `end_address` values, not the `function_name` value. The function name merely provides a hint for human auditing and debugging of the tool and the configuration file, because a developer can arbitrarily place calls to STEMv1 in source code. The `tagnote` value provides a comment on the particular slice. The `isr_key` value contains the ISR key for that slice. Note that ISR keys are mapped to slices; there is not a 1-to-1 mapping between the application and an ISR key, as in previous work [KKP03].

The configuration file is used to both randomize and derandomize the appropriate slice. Although STEMv1 takes care of derandomization, we needed to invent a tool that *randomized* slices. We built the `objrand` tool, an adaptation of the GNU `objcopy` utility. The tool is given the name of the binary to randomize (a fully compiled and statically linked executable) and the name of the configuration file. This tool reads in the SliceKeyTuples and randomizes each slice, producing a new copy of the binary. A patch is available at our website[6] under the terms of the GNU GPL. Briefly, the tool reads each section (*e.g.,* .text, .bss, .data) of the binary, tests if it contains code, and then uses the appropriate SliceKeyTuple to XOR the instruction bytes with the supplied ISR key. We added a command line configuration parameter `--isr-conf <file>` and wrapped the modified `objcopy` tool with a shell script. Running the tool on a small sample binary produces the output shown in Figure 4.5.

STEMv1 assumes a threat model that closely matches that of previous ISR efforts. Specifically, we assume that an attacker does not have access to the randomized binary or the keys used to achieve this randomization. These objects are usually stored on a system's disk or in system memory; we assume the attacker does not have local access to these resources. In addition, the attacker's intent is to inject code into a running process and thereby gain control over the process by virtue of the injected instructions. ISR is especially effective against these types of threats because it interferes with an attacker's

[6] `http://www1.cs.columbia.edu/~locasto/research/stem/objcopy.objrand.patch`

```
[michael@xoren test]$ ../objrand/bin/objrand
usage: ../objrand/bin/objrand <executable_image> [configfile]
[michael@xoren test]$ ../objrand/bin/objrand a10 a10-objrand.conf
section 0 (16/.init) vm addr is: 080480d4
section 0 (16/.init) randomized 0 bytes
section 1 (17/.text) vm addr is: 08048100
address 0x8048262 found in [0x8048262, 0x8048270] with key a7
address 0x8048263 found in [0x8048262, 0x8048270] with key a7
...
address 0x8048270 found in [0x8048262, 0x8048270] with key a7
section 1 (17/.text) randomized 15 bytes
section 2 (18/__libc_freeres_fn) vm addr is: 080c90c0
section 2 (18/__libc_freeres_fn) randomized 0 bytes
section 3 (19/__libc_thread_freeres_fn) vm addr is: 080c9908
section 3 (19/__libc_thread_freeres_fn) randomized 0 bytes
section 4 (20/.fini) vm addr is: 080c99c0
section 4 (20/.fini) randomized 0 bytes
[michael@xoren test]$
```

Figure 4.5: *A sample* `objrand` *run.* The tool considers each section of the ELF image in turn and reports how many bytes it randomized; sections that do not contain code are skipped. When the tool finds an address within the range of a SliceKeyTuple from the configuration file, it copies the randomized (in this case, randomization is equivalent to the XOR of the original value with the ISR key `0xA7`) value to the new binary image.

ability to automate the attack. The entire target population executes binaries encoded under keys unique to each instance. A successful breach on one machine does not weaken the security of other target hosts.

4.2.4 Derandomization and Detection

STEMv1 takes control of a process's execution when the `emulate_begin` function executes. From this point on, the hardware CPU is executing the instructions that belong to STEMv1, the instructions of `emulate_begin`, its core processing loop, and its helper functions. STEM's virtual CPU derandomizes and executes the instructions of the monitored slice. In order to properly derandomize a slice, STEMv1 first reads the SliceKeyTuple configuration file if this is the first time that `emulate_begin` has been executed. After loading

the configuration data, `emulate_begin` scans the slice, derandomizing as it proceeds, to locate the address of `emulate_end`[7]. After this initial scan, it then proceeds to fetch, derandomize, decode, and execute each instruction in the slice. When `emulate_begin` encounters the CALL to `emulate_end`, it stops emulation and copies the state of the virtual CPU to the hardware CPU. Execution returns to the hardware CPU[8]. STEM also relinquishes control upon the occurrence of system calls and regains control after the system call completes.

STEMv1 detects binary code injection attacks with its ISR sensor. Injected binary code will fail to derandomize correctly, producing a basic hardware exception like an illegal memory reference (manifested through the OS SIGSEGV signal) or illegal opcode. STEMv1 catches this signal and performs basic error virtualization [SLBK05] as a self–healing strategy. It optionally performs memory forensics to capture an image of the data causing the fault and forwards that data to a FLIPS instance. See Chapter 6 for more detail on FLIPS. See Chapter 8 for more information and updated experiments on the efficacy of vanilla error virtualization. A definition and explanation of error virtualization is provided in Section 2.2.1 of Chapter 2, and we show an example in Figure 3.4. Briefly, the main assumption underlying error virtualization is that a mapping can be created between the set of errors that *could* occur during a program's execution and the limited set of errors that the program code explicitly handles. By virtualizing errors, an application can continue execution through a fault or exploited vulnerability by nullifying its effects and using a manufactured return value for the function where the fault occurred. In STEMv1, the developer determines these return values by source code analysis on the return type of the offending function. Vanilla error virtualization seems to work best with server applications: applications that typically have a request processing loop that can presumably tolerate minor errors in a particular trace of the loop.

[7]This is an empty void function simply used as a placeholder or flag to indicate the end of a slice.

[8]Although we distinguish between STEM's virtual CPU (as supplied by a subset of Bochs) and the "hardware" CPU, STEM could very well run inside a virtual machine or other emulation environment, in which case the "hardware" CPU could be a piece of software.

4.3 Microspeculation Using Binary Rewriting: STEMv2

Although our first version of STEM helped demonstrate the feasibility of the basic concept of self–healing, it was ultimately unsuitable for protecting and repairing legacy systems, production software applications, and COTS (Commercial Off–The–Shelf) software. The key challenge is to apply a fix inline (*i.e.*, as the application experiences an attack) without restarting, recompiling, or replacing the process. Executing through a fault in this fashion involves overcoming several obstacles.

Besides the practical issue of not modifying source code, a self–healing mechanism must know both when to engage and how to guide a repair. In STEMv1, the developer accomplishes the former by explicitly modifying the source code (guided by the slice selection strategies we discussed in Section 4.1), while the latter is done through static error virtualization heuristics. We discuss some of the limitations of doing so in Section 2.2.1. Briefly, error virtualization alone is not appropriate for all functions and applications, especially if the function is not idempotent or if the application makes scientific or financial calculations or includes authentication & authorization checks (where the return value may be used as part of a critical calculation that is not part of error handling code). Finally, error handling often involves some minor adjustments of global state in addition to setting a return value.

We redesigned STEM's core mechanisms to improve its capabilities along two lines. We aim to provide a practical form of speculative execution for automated defense. First, we refrain from making changes to the application's source code by supervising execution with dynamic binary rewriting as supplied by the Pin [LCM+05] framework. Second, we provide a mechanism that helps maintain the semantics of program execution as closely as possible to the original intent of the application's author. STEMv2 interprets *repair policy* to guide the semantics of the healing process. Repair policy also dictates a supervision coverage policy; because it is specified externally (with respect to the source code), we can change the coverage policy at runtime. This capability is a novel method for dynamically hardening aspects of the application's execution. Finally, integrating new intrusion or fault sensors into STEMv2 is now a fairly straightforward engineering exercise.

One of our primary technical contributions is to make STEM applicable in situations where source code is not available or cannot be modified. We next review the technical

details of our design and implementation of STEMv2 (for simplicity, we simply refer to it as "STEM" in the following text). We built STEMv2 as a tool for the IA-32 binary rewriting Pin [LCM+05] framework. Figure 4.1 displays an overview of STEM's relationship to Pin. Without reference to Pin, STEM is a program analysis tool that intercepts program execution at instruction–level granularity to check a set of conditions and enforce a set of constraints on important data items if those conditions fail.

4.3.1 Core Design

Pin can most easily be thought of as a compiler that retroactively inserts new or third–party code into an existing binary at runtime. Pin provides an API that exposes a number of ways to instrument a program during runtime, both statically (as a binary image is loaded) and dynamically (as each instruction, basic block, or procedure is executed). Pin tools contain two basic types of functions: (1) instrumentation functions and (2) analysis functions. When a Pin tool starts up, it registers instrumentation functions that serve as callbacks for when Pin recognizes an event or portion of program execution that the tool is interested in (*e.g.*, instruction execution, basic block entrance or exit, *etc.*). The instrumentation functions then employ the Pin API to insert calls to their corresponding analysis functions. Analysis functions then become part of the monitored program's regular flow of execution. We note that analysis functions are invoked every time the corresponding code slice is executed; instrumentation functions are executed only the first time that Pin encounters the code slice.

STEM treats each function as a transaction. Each "transaction" that should be supervised (according to policy) is speculatively executed. In order to do so, STEM uses Pin to instrument program execution at four points: function entry (*i.e.*, immediately before a CALL instruction), function exit (*i.e.*, between a LEAVE and RET instruction), immediately before the instruction *after* a RET executes, and for each instruction of a supervised function that writes to memory. The main idea is that STEM inserts instrumentation at both the start and end of each transaction to save state and check for errors, respectively. If microspeculation of the transaction encounters any errors (such as an attack or other fault), then the instrumentation at the end of the transaction invokes the appropriate diagnosis

and repair actions in accordance with the repair policy.

STEM primarily uses the "Routine" hooks provided by Pin. When Pin encounters a function that it has not yet instrumented, it invokes the callback instrumentation function that STEM registered. The instrumentation function injects calls to four analysis routines:

1. `STEM_Preamble()` – executed at the beginning of each function.
2. `STEM_Epilogue()` – executed before a RET instruction
3. `SuperviseInstruction()` – executed before each instruction of a supervised function
4. `RecordPreMemWrite()` – executed before each instruction of a supervised function that writes to memory

STEM's instrumentation functions also intercept some system calls to support the "CoSAK" supervision policy (briefly discussed in Section 4.1) and I/O with external entities. As a reminder, CoSAK provides a statistical measure of the "vulnerability" of a particular code location; that study found most vulnerabilities lie within six functions calls of an input point.

4.3.2 Supervision Coverage Policy

One important implementation tradeoff is whether the decision to supervise a function is made at injection time (*i.e.*, during the instrumentation function) or at analysis time (*i.e.*, during an analysis routine). Consulting policy and making a decision in the latter (as the current implementation does) allows STEM to change the coverage supervision policy (that is, the set of functions it monitors) during runtime rather than needing to restart the application. Making the decision during injection time is possible, but not for all routines (because the decision to call some routines depends on runtime information which is not yet available). Since the policy decision is made only once, the set of functions that STEM can instrument is not dynamically adjustable unless the application is restarted, or unless Pin removes all instrumentation and invokes instrumentation for each function again.

Therefore, each injected analysis routine determines dynamically if it should actually be supervising the current function. STEM instructs Pin to instrument *all* functions

— a STEM analysis routine needs to gain control, even if just long enough to determine it should not supervise a particular function. The analysis routines invoke STEM's `ShouldSuperviseRoutine()` function to check the current supervision coverage policy in effect. Supervision coverage policies dictate which subset of an application's functions STEM should protect. These policies include:

- NONE — no function should be microspeculated
- ALL — all functions should be microspeculated
- RANDOM — a random subset should be microspeculated (the percentage is controlled by a configuration parameter)
- COSAK — all functions within a call stack depth of six from an input system call (*e.g.*, sys_read()) should be microspeculated.
- LIST — functions are specified by the repair policy

In order to support the COSAK coverage policy, STEM maintains a `cosak_depth` variable via four operations: check, reset, increment, and decrement[9]. Every time an input system call is encountered, the variable is reset to zero. The variable is checked during `ShouldSuperviseRoutine()` if the coverage policy is set to COSAK. The variable is incremented every time a new routine is entered during `STEM_Preamble()` and decremented during `STEM_Epilogue()`.

4.3.3 STEM Operation

Although STEM can supervise an application from startup, STEM benefits from using Pin because Pin can attach to a running application. For example, if a network sensor detects anomalous data aimed at a web server, STEM can attach to the web server process to protect it while that data is being processed. In this way, applications can avoid the startup costs

[9]The procedure that we describe for maintaining this variable is actually a rough approximation of the "distance" from a system call. It is more appropriately described as the distance *following* a system call, because we cannot reliably detect the distance to the *next* system call unless we keep a static CFG or profiles or cause another thread of execution to explore the paths in front of us.

involved in instrumenting shared library loading, and can also avoid the overhead of the policy check for most normal input.

STEM starts by reading its configuration file, reading the repair policy file, attaching some command and control functionality (described in Section 4.3.4), and then registering a callback to instrument each new function that it encounters. STEM's basic algorithm is distributed over the four main analysis routines.

4.3.3.1 Memory Log

Since STEM needs to treat each function as a transaction, undoing the effects of a speculated transaction requires that STEM keep a log of changes made to memory during the transaction. The memory log is maintained by three functions: one that records the "old" memory value, one that inserts a marker into the memory log, and one that rolls back the memory log and optionally restores the "old" values. STEM inserts a call to `RecordPreMemWrite()` before an instruction that writes to memory. Pin determines the size of the write, so this analysis function can save the appropriate amount of data. Memory writes are only recorded for functions that should be supervised according to coverage policy. During `STEM_Preamble()`, Pin inserts a call to `InsertMemLogMarker()` to delimit a new function instance. This marker indicates that the last memory log maintenance function, `UnrollMemoryLog()`, should stop rollback after it encounters the marker. The rollback function deletes the entries in the memory log to make efficient use of the process's memory space. This function can also restore the "old" values stored in the memory log in preparation for repair.

4.3.3.2 STEM_Preamble()

This analysis routine performs basic record keeping. It increments the COSAK depth variable and maintains other statistics (dynamic number of routine instances supervised, *etc.*). Its most important tasks are to (1) check if supervision coverage policy should be reloaded and (2) insert a function name marker into the memory log if the current function should be supervised.

4.3.3.3 STEM_Epilogue()

STEM invokes this analysis routine immediately before a RET instruction. Besides doing its part to maintain the COSAK depth variable, this analysis routine ensures that the application has a chance to self-heal before a transaction is completed. If the current function is being supervised, this routine interprets the application's repair policy (a form of integrity policy based on extensions to the Clark-Wilson integrity model: see Chapter 5 for details). If the repair succeeds or no repair is needed, then STEM commits the transaction. If not, and an error *has* occurred, then STEM falls back to crashing the process (the current state of the art) and reporting the alert.

This analysis routine delegates the setup of error virtualization to the repair procedure. The repair procedure takes the function name, current architectural context (*i.e.*, CPU register values), and a flag as input. The flag serves as an indication to the repair procedure to choose between normal cleanup or a "self-healing" cleanup. While normal cleanup always proceeds from `STEM_Epilogue()`, a self-healing cleanup can be invoked synchronously from `STEM_Epilogue()` or asynchronously from a signal handler. The latter case usually occurs when STEM employs a detector that causes a signal such as SIGSEGV to occur when it senses an attack. At this point, STEMv2 can perform some memory forensics in a fashion similar to STEMv1 to forward a snapshot of memory state to FLIPS; doing so supports automatically generating an exploit signature to maintain an external integrity posture.

Normal cleanup simply entails deleting the entries for the current function from the memory log. If self-healing is needed and the repair policy indicates that memory rollback should occur, then the values from the memory log are restored. In addition, a flag is set indicating that the process should undergo error virtualization according to the repair policy, and the current function name is recorded.

4.3.3.4 SuperviseInstruction()

The job of this analysis routine is to intercept the instruction that immediately follows a RET instruction. By doing so, STEM allows the RET instruction to operate as it needs to on the architectural state (and by extension, the process stack). After RET has been invoked, if the flag for repair and error virtualization is set, then STEM looks up the appropriate

error virtualization value according to policy. STEM then performs error virtualization by adjusting the value of the `%eax` register and resets the error virtualization flag. STEM next invokes the Ripple interpreter and RealPaver. STEM sets a collection of memory locations to values indicated by the constraint solver. STEM then ensures that the function returns appropriately by comparing the return address with the saved value of the instruction pointer immediately after the corresponding CALL instruction. Finally, STEM calls the Pin function `PIN_ExecuteAt()` with the updated architectural context.

4.3.4 Additional Controls

STEM includes a variety of control functionality that assists the core analysis routines. STEM defines three signal handlers and registers them with Pin. The first sets a flag indicating that policy and configuration should be reloaded, although the actual reload takes place during the execution of the next `STEM_Preamble()`. The second signal handler prints runtime debugging information. The third intercepts SIGSEGV (for cases where detectors alert on memory errors, such as address space randomization). That handler invokes the repair procedure.

STEM supports a variety of detection mechanisms, and it uses them to measure the integrity of the computation at various points during program execution and set a flag that indicates `STEM_Epilogue()` should initiate a self-healing response. Our current set of detectors includes the integrity conditions defined by the repair policy, a basic function call anomaly detector, and a shadow stack that detects integrity violations of the return address or other stack frame information. STEM also intercepts a SIGSEGV produced by an OS that employs address space randomization [BDS03].

STEM also provides a mechanism to capture aspects of an application's behavior. This profile can be employed for three purposes: (a) as input to an anomaly sensor to detect application misbehavior, (b) to aid self-healing, and (c) to validate the self-healing response and ensure that the application does not deviate further from its known behavior. STEM can capture aspects of both control flow (via the execution context) and data flow (via function argument and return values).

4.4 Summary

Both versions of STEM provide microspeculation as a service to software applications. STEMv2 does so in a fashion that makes it applicable to COTS software: it does not require source code modifications, and it imposes a roughly 2X performance (see Chapter 8) penalty on normal execution for whole-application supervision (compared to 30X for STEMv1 [SLBK05]). STEMv2 also provides a mechanism to interpret repair policy to assist the healing process and improve the semantic correctness of the repair. STEMv2, however, does not provide an ISR sensor, as Pin's decoding routines would need to be modified.

Using STEMv2 to supervise dynamically linked applications directly from startup incurs a significant performance penalty (as shown in Table 8.3), especially for short-lived applications. Most of the work done during application startup simply loads and resolves libraries. This type of code is usually executed only once, and it probably does not require protection. Even though it may be acceptable to amortize the cost of startup over the lifetime of the application, we can work around the startup performance penalty by employing some combination of three reasonable measures: (1) statically linking applications, (2) only attaching STEM after the application has already started, (3) delay attaching until the system observes an IDS alert. We evaluate the second option by attaching STEMv2 to Apache after Apache finishes loading. Our results (shown in Table 8.4) indicate that Apache experiences about a 1X performance degradation under STEMv2. More details on workload distribution and performance characteristics of STEMv1 and STEMv2 can be found in Chapter 8.

Chapter 5

Maintaining an Internal Integrity Posture

The essence of maintaining an internal integrity posture is the continuous and automatic detection, diagnosis, and repair of violated integrity constraints on critical data items within a system. In order to accomplish this task, we need both a runtime supervision environment (described in Chapter 4) and a way of controlling this supervision process. The remainder of this chapter describes the novel concept of *repair policy*, including a model, language, and interpreter for the policy.

We also include a brief survey of assertion use in a variety of applications. This cursory examination lends some support to the belief that important assertion conditions involve a relatively small amount of state. This result is encouraging because it implies that repair constraints can be relatively small, easy to understand, and quickly satisfied, although detecting inconsistencies is not the same as knowing how to fix them.

5.1 Motivation

Many systems contain facilities for dealing with signals about error conditions. We advocate augmenting this paradigm with *affirmative* techniques that continuously constrain process behavior rather than attempt to handle an unanticipated error at an arbitrary point in its control flow. Specifically, we propose leveraging the Clark-Wilson Integrity Model to

express positive statements about program behavior in the form of boolean conditions on program state and execution artifacts. This collection of assertions defines a policy useful for continuously checking a process's behavior. Not only can such assertions help detect behavior deviations, but they also provide a goal state for automated repair based on constraint satisfaction. These Self-Correcting Assertions[1] (SelCA) relieve a programmer of the unreasonable burden of constructing incomplete or unsound exception handling code *for unanticipated failure conditions*.

Exception handling helps address anticipated faults (*e.g.,* the OS cannot open a file). However, attacks and unanticipated faults are inherently open-ended and continuously evolving. Exceptions manifest as little more than a signal that describes the symptom experienced, rather than identifying the root cause of the problem. Indeed, exceptions contain little *programmatic* support for discovering this type of information[2]. There is no technical reason that this type of access cannot exist, and certain well–designed exception objects have a number of such methods. Each is quite specialized, however, and the standard exception API (at least in Java) is quite anemic, essentially containing functionality only useful for logging the exception. The only restriction on providing this type of access is that the set of interesting information is *too* open-ended. Some exception handling systems provide access to the stack state as part of the guaranteed exception object state, but program language designers often leave it to the application author to decide how to touch process memory during error conditions. We encourage efforts to augment exceptions with programmatic access to data that can assist efforts to self–heal. For example, the Java exception signaling an out of bounds array index reference does not currently provide access to the actual index value (although it allows the programmer to *set* such a value by providing an argument to the exception's constructor, no corresponding accessor is documented), the Object being accessed, the data to transfer, and other state information that would be

[1]Note that we do not change the content of an assertion, but rather the values bound to the symbols in the assertion. Having detected a violated condition (*e.g.,* $x < 3$), we do not alter the actual asserted condition (*e.g.,* $x >= 3$); rather, we change the value of the atoms (*e.g.,* set x to 2).

[2]By "programmatic" access, we mean standard methods for retrieving, generating, or repairing critical data objects in the syntax and semantics of the source language of the system or application.

helpful during a diagnosis step.

Thus, even when a programmer employs exception handling, his awareness of the exact state of the system and the violated condition (rather, his lack thereof) hampers its use as a recovery mechanism. The programmer's awareness of the error begins at the moment that the exception is caught: a point in the control flow that may exist in a completely different scope and level of abstraction. In these cases, the exception would need to marshal locally scoped data into and out of the exception instance.

```
try {
    int x = Integer.parseInt(s);
} catch(NumberFormatException n) {
    log("unexpected x value, setting to default");
    x = DEFAULT_X_VALUE;
    s = ""+x;
}
```

Figure 5.1: *Handling a "simple" exception.* Even this basic exception requires a moderate amount of infrastructure and design decisions from a developer.

Even the example shown in Figure 5.1 requires a non-trivial amount of planning. A developer must know that `parseInt()` could throw an exception, and that `s`'s content comes from an untrusted source (he might otherwise catch and ignore the exception). A default value for `x` must exist. Furthermore, the statements in the handler reveal dependencies on the rest of the code base. The developer must know the syntax and semantics of the logging infrastructure and must create an informative error message. Finally, he must consider whether or not to "fix" the input or let other code handle the illegal value. Even for this basic violation, the programmer faces a considerable challenge in formulating the best recovery strategy. He may not be sure how far the system has drifted off course after a number of functions and statements in the same scope as the exception have already executed.

5.1.1 Exception Shortcomings

We consider exceptions of limited use in software self–defense systems for the following reasons:

1. Exceptions increase execution complexity by reducing certainty as to where they will be handled.

2. Expressing exceptions during specification of the "success path" in a piece of code seems counterintuitive. Even when compilers force a programmer to employ them, the programmer can usually avoid specifying a meaningful handler. He takes no coherent diagnosis step because thinking about how the system may fail during attempts to specify how it should succeed is a difficult mental exercise.

3. Even when a programmer employs them, he may have an inexact notion of the system's state. Since he cannot enumerate all possible "error states", he cannot enumerate all responses, express them in code, specify a module to choose among them, or construct a test harness to verify them.

Ideally, software applications should actively diagnose and repair themselves when faced with errors and malicious events. Even though the need to proactively anticipate known error conditions can justify the use of exception handling, systems require a complementary mechanism for reactively detecting and repairing unanticipated faults and attacks. Self–healing also requires *affirmative* mechanisms that continuously constrain process behavior, detect violations, construct appropriate state changes to repair the violation, and verify that the applied repair worked. *Error virtualization* (discussed in Section 2.2.1) is one such self–healing strategy, but this technique is oblivious to the semantics of the code it "heals." The central research challenge, then, is to provide affirmative techniques that can account for the semantics of code in need of repair.

5.1.2 Problem: Semantics–Oblivious Healing

Achieving a semantically correct response remains a key problem for self–healing techniques. Self–healing strategies that execute through a fault by effectively pretending it can be handled by the program code or other instrumentation may give rise to semantically incorrect responses. In effect, naïve self–healing may provide a cure worse than the disease. Figure 5.2 illustrates a specific example: an error may exist in a routine that determines the access control rights for a client. If this fault is exploited, a self-healing technique like error

virtualization may return a value that allows the authentication check to succeed. This situation occurs precisely because the recovery mechanism is oblivious to the semantics of the code it protects.

```
int login(UCRED creds)
{
     int authenticated = check_credentials(creds);
     if(authenticated) return login_continue();
     else return login_reject();
}
int check_credentials(UCRED credentials)
{
     strcpy(uname, credentials.username);
     return checkpassword(lookup(uname), credentials);
}
```

Figure 5.2: *Semantically Incorrect Response.* If an error arising from a vulnerability in `check_credentials` occurs, a self-healing mechanism may attempt to return a simulated error code from `check_credentials`. Any value other than `0` that gets stored in `authenticated` causes a successful login. What may have been a simple DoS vulnerability has been transformed into a valid login session by virtue of the "security" measures. STEM interprets *repair policy* to intelligently constrain return values and other application data.

5.1.3 Seeds of a Solution

We require a solution that can be transparently applied to a system and has the ability to maintain the semantic correctness of the protected application. This solution involves identifying instrumentation locations and creating the actual repair mechanism. We take our inspiration from two basic techniques (neither of which is entirely appropriate on its own or in raw combination). Both of the following options have serious shortcomings; they only provide the basis of our inspiration and illustrate the evolution of our thinking. The first technique relies on annotating the source code to (a) indicate which routines should or should not be "healed" and (b) provide appropriate return values for sensitive functions. We find this technique unappealing because of the need to modify source code.

As another way of instrumenting the source, we could encode these annotations by

overriding the default behavior of the common `assert` macro. This primitive evaluates the logical condition supplied as its argument. If the condition evaluates to false, the routine reports an error and subsequently aborts execution. In order to use `assert` routines for self–healing, we could change the implementation to adjust the values of the relevant program state (*i.e.,* state involved in the condition supplied to the routine) to make the asserted expression true, thereby allowing `assert` routines to repair as well as detect and notify. We could convert existing `assert` macros via runtime instrumentation or by altering the compiler or preprocessor.

Unfortunately, this approach suffers from a few deficiencies, aside from the issue that some legacy systems lack assertions altogether. First, `assert` routines most likely exist as a last resort to crash the program if the asserted condition fails. Changing these semantics via overloading interferes with that goal. Second, `assert` macros evaluate the supplied expression. These semantics are problematic if the evaluated condition has side effects[3]. Even though directly overloading vanilla assertions does not seem suitable, the intuition seems sound: assertion conditions can provide a useful primitive for self–healing and automated repair. A designer, developer, tester, or administrator can easily express constraints critical to continued correct operation. Finally, constraints also provide a good indication of the complexity of automated repair, and Chapter 8 presents a survey of assertion use in a variety of applications.

Since source-level annotations serve as a vestigial policy, we articulate a way to augment self–healing approaches like error virtualization with the notion of *repair policy*. A repair policy (or a recovery policy — we use the terms interchangeably) is specified separately from the source code and describes how execution integrity should be maintained after an attack or failure is detected. Repair policy can provide a way for a user to customize an application's response to an intrusion as a way to achieve an automated recovery.

The problem of creating a repair policy system is really a problem of policy model

[3]The `assert` manual page notes: "`assert()` is implemented as a macro; if the expression tested has side-effects, program behavior will be different depending on whether NDEBUG is defined. This may create Heisenbugs [ed. Heisenbugs are errors that exactly express Heisenburg's Uncertainty Principle: they cannot be simultaneously observed and measured with any precision.] which go away when debugging is turned on."

design. Repair policies must be expressed in a language that contains the primitives needed to express reasonably useful prescriptions that help achieve a semantically correct response. In general, a language requires a model of computation, a syntax, and a semantics. The concept of repair policy is essentially about integrity, so we look toward the Clark-Wilson Integrity Model (described in Chapter 3) to help inform our choice of syntax, semantics, and runtime environment.

5.2 An Integrity Repair Model

We provide a model of computation and derive some syntactic constructs for repair policy by extending the Clark-Wilson Integrity Model (CW) [CW87] to include the concepts of (*a*) repair and (*b*) repair validation. CW is ideally suited to the problem of detecting when constraints on a system's behavior and information structures have been violated. The CW model defines rules that govern three major constructs: constrained data items (CDI), transformation procedures (TP), and integrity verification procedures (IVP). An information system is composed of a set of TPs that transition CDIs from one valid state to another. The system also includes IVPs that measure the integrity of the CDIs at various points of execution.

Although a TP should move the system from one valid state to the next, it may fail for a number of reasons (incorrect specification, a vulnerability, hardware faults, *etc.*). The purpose of an IVP is to detect and record this failure. CW does not address the task of returning the system to a valid state or formalize procedures that *restore* integrity. In contrast, repair policy focuses on ways to recover after an unauthorized modification. Our extensions supplement the CW model with primitives and rules for recovering from a policy violation and validating that the recovery was successful.

The straightforward nature of the CW model does not require a great deal of added complexity to increase its power to describe the repair of failed integrity constraints. Our extensions to the model include two new types of procedures: Repair Procedures (RP) and Repair Validation Procedures (RVP). These additions help define *repair policies*. These policies specify a goal state; that is, they describe what changes to the system must occur

in order for a set of IVPs to measure some set of CDIs and return "VALID."

Our additions are based on four new rules that supply repair and repair validation procedures, although the **C7** rule can easily be combined with rule **C3** (see Chapter 3).

1. **C6:** All RPs and RVPs must be certified valid. For RPs, certification indicates that it returns a given set of CDIs to a valid state. For RVPs, certification mirrors that of rule **C1**. For each RP and RVP and each set of CDIs that they may manipulate, the security officer must specify a "relation" which defines that execution. A relation is of the form: $(RP_i \vee RVP_i, (CDI_a, CDI_b, CDI_c, \ldots))$, where the list of CDIs defines a particular set of arguments for which the RP or RVP has been certified.
2. **E5:** The system must maintain a list of relations of the form: (USERID, RP_i, $(CDI_a, CDI_b, CDI_c, \ldots)$ which relates a user, a RP, and the CDIs that a RP may adjust back to a satisfactory value assignment.
3. **E6:** The system must maintain a list of relations of the form: (USERID, RVP_i, $(CDI_a, CDI_b, CDI_c, \ldots)$ which relates a user, a RVP, and the CDIs that a RVP may measure to make sure they have a satisfactory value assignment.
4. **C7:** The list of relations in **E5** and **E6** must be certified to meet the separation of duty requirements. For example, an RVP must be implemented and certified by a user other than the user that certified the corresponding IVP and RP. This SoD is useful in case either of these procedures was maliciously or incorrectly certified.

In the example of Figure 5.2, variables like `authenticated`, `uname`, and `auth_OK` are CDIs. The `login`, `strcpy`, and `check_credentials` functions are TPs. Since manipulation of `uname` may overwrite a stack return address, any standard detection mechanism for this type of exploit serves as an IVP for the CDIs of the stack frame. Likewise, IVPs for the bounds of `uname` can detect violations. If the IVP indicates that the TP has failed to maintain the integrity of this CDI, then a RP is needed that will set an appropriate value (presumably FALSE or reject) for the CDIs `auth_OK` and `authenticated` as well as additional CDIs. Furthermore, a RVP should be invoked to test the results of this repair. An RVP can help ensure that the value of the CDIs are not simply set to a satisfying or legal value, but rather the most appropriate value given the current system state.

One of the key implications of these extensions is that error handling is associated with a CDI, not with a point in the application's control flow (as is the case with exceptions).

Associating an exception with control flow has the potential to lead to convoluted error handling, with an exception being handled at a remote place in the source from where the error occurred and at a different level of abstraction. In contrast, our model does not impose a direct relationship between RPs and TPs. Rather, an indirect mapping exists through the CDIs that form the relations of **E2** and **E5**. Consequently, we leave as a policy specification choice which TPs a particular IVP should be invoked for, and this situation naturally reflects the choice of a detection mechanism and how often it is executed.

Strategies range from random selection to full coverage (invoking an IVP after each TP). Other interesting enforcement choices include selection based on proximity of a TP to input handling routines or the error history of a TP. For example, StackGuard and related approaches [CPM+98, Eto00] attempt to detect changes to the return address (or surrounding data items) of a stack frame. If the integrity of these values is violated, execution is halted. These approaches quite naturally fit the Clark-Wilson Integrity Model: the CDI is the return address in the current frame, the TP is the procedure for popping a stack frame, and the IVP is the procedure by which these systems attempt to verify that the CDI is still valid. What these systems lack is the notion of both RPs and RVPs. They have a vestigial RP: optionally logging the fault and then terminating the process.

In the literature, examples of "complete" protection include the techniques advocated by StackGuard (at least with respect to a particular type of CDI). Similarly, array-bounds checking in Java ensures each array access falls within the limits of the array. Complete process emulation is even more expensive and employed in approaches like Instruction Set Randomization [BAF+03, KKP03] and binary tainted dataflow analysis [CCCR05, NS05]. Taint analysis is equivalent to tracking Unconstrained Data Items (UDI) through the system until they affect the control flow of a TP. The VSEF [NBS06] system offers a workaround, where only the portions of a process's instruction stream that are confirmed to be vulnerable to a code injection attack are "hardened" (although the underlying tool that VSEF uses still requires the rest of the instruction stream to be emulated, but without instrumentation).

RVPs are associated with a set of CDIs and serve to check the validity of the CDIs after the application of an RP. Figure 5.2 expresses the arrangement of TP, IVP, RP, and RVP as a form of an Inline Reference Monitor. The model also frees us from having to modify

```
reference_monitor(TP t, CDIList c)
     IVP ivp = lookupIVP(t,c);
     boolean valid = ivp(c);
          if(!valid)
                RP rp = lookupRP(c);
                RVP rvp = lookupRVP(c);
                rp.execute(c);
                boolean verified = rvp(c);
                if(!verified)
                      abort();
     return;
```

Figure 5.3: *Integrity Repair Model Pseudocode.* The IRM can fall back to aborting execution if the repair procedure does not succeed in restoring integrity to the CDI set.

the source code; the invocation of the IRM can be inserted in a program binary.

5.3 Ripple: A Repair Policy Language

Ripple is a domain–specific language that we designed to express repair policy. We have integrated a Ripple policy interpreter into STEMv2. In this section, we briefly cover the important aspects of Ripple, the execution of policies written in it, and its use in STEMv2. We give a more in-depth treatment of the Ripple Language in Appendix A.

We designed Ripple as a domain–specific language in order to have well–defined operations and syntax specific to solving the problems in the space with a few easy keystrokes (as opposed to large chunks of C/C++ or Java code). Ripple can therefore be reused across problems and systems in this area and provide a common language (*i.e.*, syntax, semantics, and idioms) for people charged with solving these problems.

Ripple is, however, quite powerful and enjoys direct access to the underlying hardware settings via the binary supervision environment. Therefore, a certain amount of care should be used despite the safety features and convenience. For example, with Ripple, wiping out the entire memory address space of a supervised process can be accomplished with a single line of code.

5.3.1 Repair Policy Overview

STEM interprets repair policy to provide a mechanism that can be selectively enforced and retrofitted to the protected application without modifying its source code (although *mapping* constraints to source level objects assists in maintaining application semantics). As with most self–healing systems, we expect the repairs offered by this "behavior firewall" to be temporary constraints on program behavior — emergency fixes that await a more comprehensive patch from the vendor.

Repair policy is specified in a file external to the source code of the protected application and is used only by STEM (*i.e.*, the compiler, the linker, and the OS are not involved). This file contains a Ripple specification and describes the legal settings for variables in an aborted transaction. The basis of the policy is a list of relations between a transaction and the CDIs that need to be adjusted after a policy violation, including the return address and return value. Complete repair policy is a wide-ranging topic; in this thesis we consider a simple form that:

1. specifies appropriate error virtualization settings to avoid an incorrect return value that causes problems like the one illustrated in Figure 5.2
2. provides memory rollback for an aborted transaction
3. sets memory locations to particular values

Repair policy has a number of benefits. There is no need to modify an application's source code (and thus no need to recompile). Repair policies allow us to specify valid error virtualization values and constrain any other needed state cleanup. In addition, repair policy can help address newly discovered vulnerabilities (such as those revealed during fuzz testing by a vendor or security professional) without the need for an explicit patch to the source. In much the same way that the Shield [WGSZ04] project proposes inserting vulnerability–specific network filters into the protocol stack so as to avoid modifying the application, we can generate and distribute repair policies for newly discovered vulnerabilities without requiring an invasive patch. Finally, an incorrectly specified repair policy need not negatively impact the operation of the application; an administrator can "turn off"

a broken repair policy without affecting the execution of the program — unlike currently deployed binary patch technology like Windows Update.

```
symval AUTHENTICATION_FAILURE = 0;

ivp MeasureStack :=:
   ('raddress=='shadowstack[0]);

rp FixAuth :=:
   ('rvalue==AUTHENTICATION_FAILURE),
   (unroll);

tp check_credentials
   &MeasureStack
   :=:
   &FixAuth;
```

Figure 5.4: *Sample Repair Policy.* If the TP named `check_credentials` fails, then the memory changes made during this routine are reset and STEM stores the value `0` in the return value (and thus into `authenticated`), causing the login attempt to fail.

Figure 5.4 shows a sample policy for our running example. The first statement defines a symbolic value. The latter three statements define an IVP, RP, and TP. The IVP defines a simple detector that utilizes STEM's shadow stack. The RP sets the return value to a semantically correct value and indicates that memory changes should be undone, and the TP definition links these measurement and repair activities together. An RP can contain a list of asserted conditions on CDIs that should be true after self–healing completes. The example illustrates the use of the special variable `'rvalue` (the apostrophe distinguishes it from any CDI named `rvalue`). This variable helps customize vanilla error virtualization to avoid problems similar to the one shown in Figure 5.2.

5.3.2 Model of Computation

Ripple's model of computation is event–driven. Control flow does not follow the Ripple program text. Instead, Ripple's execution is a grouping of assertions that correspond to a point in the control flow graph (CFG) of the supervised application. A four-tuple of

language constructs, $\{TP, IVP, RP, RVP\}$, collectively speculate, supervise, repair, and validate an execution primitive (*e.g.*, a function, procedure, routine, or method; a basic block; or a single machine instruction).

The basic flow of events is defined by the dynamic CFG of an application as it executes for a particular input. In this way, the relationships that Ripple governs can be considered a type of aspect (as in Aspect-Oriented Programming [EFB01]). The Ripple runtime environment only evaluates an IVP when a TP is encountered by the supervision environment. In this fashion, it *continuously cross-cuts* the application's execution, and the history provides a trace of Ripple's intersection with the application itself.

Ripple is predicated on an analogy between program execution and a transactional model. Our underlying assumption maintains that Ripple can treat certain portions of an application (*e.g.*, functions: although basic blocks or even individual instructions serve equally well) as a transaction that may be aborted if it does not succeed. In Ripple's case, success is defined by a combination of two things: a measurement made by a detector in the supervision framework (such as a host system call based anomaly detection component) or the failure of an IVP or set of IVPs defined by a Ripple policy.

Ripple does not contain the procedural code familiar to Java and C programmers. Instead, it describes portions of the phase space of an application and what the Ripple programmer believes to be true about program state and environment when in a particular locale of that phase space. This description is based on a collection of constraints that are bound to each type of Ripple procedure. The order of evaluation of constraints is not defined by the language execution model so that the particular implementation of the runtime Ripple interpreter is free to select the "best" constraint to satisfy first (although we note that, as explained in Chapter 3, constraint satisfaction is commutative[4]). Thus, the order of definition in a comma clause does not define an order of constraint evaluation.

While Ripple's control flow is event-driven, the recommended order of writing program statements is fixed. In particular, symbols and CDIs must be defined before they are used elsewhere in the program text. In addition, all IVPs must be declared before any TP.

[4]In other words, it does not matter for correctness which order constraints are satisfied in — although performance may improve with a fortuitous or clever order of evaluation.

Finally, any RP must be declared before a TP. In this way, there are five logical sections to a Ripple policy file. There are no "syntactic sugar" delimiters between these portions of a policy file:

1. type definition (define types for use with CDIs and UDIs)
2. data definition and mapping (CDI, UDI, and symval defs)
3. IVP definition
4. RP definition
5. RVP definition
6. TP definition

Therefore, a well-formed Ripple policy script *should* be read from top to bottom, starting with the type and variable definitions and mappings, proceeding through the detection conditions specifications, moving through the repair procedure constraint collection and the repair validation code and ending with the bindings of the previous entities with a particular TP.

If the Ripple interpreter encounters a problem during policy parsing, it will halt STEM's startup or attach procedure. If it encounters a problem during runtime (such as an illegal memory dereference) it will log an entry in STEM's log file and abort the current procedure.

5.3.3 Repair Policy Execution

The WVM is a standalone program that results from compiling the code generated by Antlr[5] from the Ripple grammar. The WVM accepts the name of a Ripple policy file on the command line, parses it to create the appropriate language data structures, opens a socket to communicate with clients of the WVM, and stays memory-resident until it receives a shutdown message. Although the WVM can be used to simply parse Ripple programs, it is not much use beyond that without being able to weave in the execution of a client

[5]A tool similar to lexer and parser generators like yacc, flex, and bison. It is available at `http://www.antlr.org/`

program. In this respect, Ripple and the WVM can be seen as computation that govern an aspect of a program's execution.

STEM uses a shim to communicate with the WVM. The shim is responsible for maintaining the data structures introduced by a repair policy on the STEM side. As STEM starts up, it reads in the Ripple policy file, uses the shim to fork off the WVM, and passes the policy to the WVM via the shim.

The WVM parses the Ripple policy and creates the appropriate data structures indicated by the policy (list of CDIs, symvals, IVPs, TPs, and RPs). It notifies its client (in this case, STEM) about these data structures. The main purpose of doing so is for STEM to associate CDIs with particular memory locations. As a result, at critical points in execution (after every TP), STEM is able to consult the WVM with the current values of the program state. If repair is needed (the IVP conditions for that TP fail to evaluate to true), the WVM invokes RealPaver to satisfy constraints in the appropriate RP. The WVM passes those CDI assignments back to STEM, which then sets the appropriate memory locations to the appropriate values, since it has direct control over the program's address space.

5.4 Assertion Use

Using assertions to continuously constrain program behavior is one of the motivations for repair policy. Although we should not and do not use assertions directly (for reasons explained in Section 5.1.3), we examine the hypothesis that satisfying "typical" boolean conditions is feasible: the time to determine a satisfiable assignment of values to expression atoms is short. We perform a survey on open source applications written in both C/C++ and Java to characterize the complexity of `assert` expressions in each application by the properties summarized in Table 5.2.

The results agree with our intuition that assertion expressions do not have a high degree of complexity. The average number of the atoms (an atom is defined as an operator and its arguments) per `assert` is less than two. Expressions rarely have 2 to 5 atoms, and we did not find more than 5 atoms in any of the analyzed applications. Most atoms' structural detail involves the equality operator (a relatively easy relation to satisfy). Assertions

Table 5.1: *Assertion Statistics*

app	*ta*	*tma*	*eqa*	*exa*	*lga*	*fna*	*tfa*	*tsa*	*cmx*	1*a*	2*a*	3*a*	4*a*	5*a*
cvs-1.11.22	245	194	133	86	53	10	4	0	1.17	216	19	8	2	0
gaim-1.5.0	5	81	1	4	0	0	0	0	1	5	0	0	0	0
httpd-1.3.36	248	53	153	30	74	57	11	0	1.10	224	24	0	0	0
james-2.2.0	115	129	84	54	17	54	1	0	1.347	95	8	7	2	3
mplayer-1.0pre8	849	166	500	95	409	107	40	0	1.214	715	96	28	10	0
openssh-4.3p2	7	69	0	0	7	0	0	0	1	7	0	0	0	0
qemu-0.8.1	120	35	81	3	33	5	34	0	1.25	100	13	4	3	0
sendmail-8.13.7	52	147	8	6	33	4	7	0	1.038	50	2	0	0	0

involving function calls do not occur with high frequency, as shown in Table 5.1. This is encouraging, because this type of expression is impossible to fix automatically in the general case. Some applications show a number of assertions that are unequivocally meant to fail; they confirm our hypothesis that using the compiler to derive repair code from existing assertions is inadvisable. Rather, programmers, users, administrators, or policy gurus should construct repair policy. We manually analyzed a few other applications (`valgrind-3.2.0` and `httpd-2.2.2`) and observed similar results: low complexity, 1-atom assertions.

The lesson here is that information critical to continued operation is not of high-complexity, resulting in small and quickly evaluated IVPs (it is unclear if these characteristics of typical assertions apply to RPs in general).

5.5 Summary

The heart of the CW model and our extensions to it are lists of relations that constrain the values of CDIs. Ripple provides a way to specify the semantics of failure recovery so that when a constraint is violated, repair activities can take place as quickly and correctly as possible. Ripple constrains the execution of automatic remediation in a customizable way.

The modified CW model serves our overall goal quite well: human involvement during repair is limited, as repair procedures are carried out automatically, although their synthesis is currently manually driven. CW provides concrete and measurable constraints that are certified to be enforceable (IVPs). CW provides separation of duty as one of its core

Table 5.2: *Key for Table 5.1*

app	the application under review
ta	total # of assertions in code
tma	total # of assertions in docs
eqa	# of equivalence assertions
exa	# of existential assertions (*e.g.*, x!=NULL)
lga	# of logical assertions (*e.g.*, $x < 5$)
fna	# of assertions involving a function call
tfa	# of assertions meant to fail (*e.g.*, 1!=1)
tsa	# of assertions meant to succeed (*e.g.*, 0==0)
cmx	complexity of assertions, measured in # of atoms
1a	total # of 1-atom assertions
2a	total # of 2-atom assertions
3a	total # of 3-atom assertions
4a	total # of 4-atom assertions
5a	total # of 5-atom assertions

constructs: integrity is easier to maintain if trust is not vested wholly in one principal. SoD breaks down the actions of a transaction so that principals must conspire to subvert it. Our examples, however, do not show the RP checking the conditions of the IVP; each RP trusts the execution of the IVP. In principle, the IVP would issue some type of self–verifying alert that each RP could check. Simple randomized schemes provide less onerous coordination and a probabilistic rate of failure for any collusion. Our modifications satisfy the requirements for repairing integrity by returning data and other execution artifacts to a valid state through the use of RPs. Context for the repair actions is captured via the set of CDIs that should be fixed, providing a precise snapshot of the relevant state. Finally, the RVP construct provides a mechanism to measure the validity of the repair operation.

5.5.1 Choosing Repair Scope

Identifying locations in program code to insert calls to the reference monitor or policy evaluator is a challenging task. Many approaches to this problem exist, ranging from complete instrumentation to selective, retroactive insertion of specific checks. This range reflects the tradeoff between the level of protection for a system and the impact on system performance. Depending on the protection mechanism, IVP invocation could occur after every machine instruction, after every basic block, or after some specific class of machine instructions (*e.g.*, control-flow transfer, arithmetic, logic, floating point, *etc.*). Locations at higher levels of abstraction are also plausible places to insert policy checks. For example, the Java policy access control mechanism relies on calls to the `AccessController` being interwoven throughout the standard classes of the Java library. The approach we take in this thesis is to remain flexible: only TPs specified in the policy are instrumented with calls to the appropriate set of IVPs.

Ganapathy *et al.* [GJJ06] suggest an interesting approach to retroactive placement of security checks (although the security checks they examine have to do with authorization policy enforcement). After a small collection of enforcement points are manually specified, their system synthesizes "fingerprints" of these code sequences and automatically identifies other locations in the program that match the fingerprint. These locations are candidates for instrumentation. In general, our approach does not require that the author of a repair policy know that a TP has a vulnerability: likely candidates may be selected according to heuristics or real evidence that a vulnerability exists, such as alerts from an IDS. Another possible approach may be to have the enforcement mechanism remember a snapshot of the application environment when attacks *do* occur: this state is useful as future evidence. Follow-on work could build a system that learns when environment conditions are ripe for attack. The recreation of those conditions could trigger the enforcement mechanism.

5.5.2 Discussion

There are some limitations inherent in the approach and other potential objections; we cover some in the sections of Chapter 3 dealing with Clark-Wilson and constraint satisfaction. We briefly cover some of these issues here as well.

The constraining relation on a CDI may be arbitrarily complex and include calls to functions. In addition, expressions may only represent a measurement of a high–level property of another data structure or CDI. We term such a relationship a "state delegate." For example, a CDI named `num_sorted` may be constrained to be less than the value `10`. Repair likely involves not only modifying the value of the CDI `num_sorted`, but also the underlying CDI (some array or data collection). This type of algorithmic repair is likely not achievable in general, although it may be possible to achieve on a case-by-case basis.

This thesis limits its focus to constraints that are expressible as boolean formulas. As discussed in Chapter 3, more complex expressions exist (*e.g.*, relations that are equivalent to a Turing Machine description) and we defer their study to future work, partly because boolean formulas are powerful enough for the purpose of expressing the constraints needed for repair, and partly because we suspect the automatic synthesis of *correct* RPs from TM-equivalent IVPs is undecidable. The model supports manual synthesis and certification of these RPs.

Some may protest that programmers must synthesize repair policy ahead of time; if so, why not write the software correctly in the first place? In cases where repair policy is synthesized by a human, SoD implies that the programmer should not specify the recovery policy; the end-user, a policy guru, or system administrator may actually provide a better specification. Furthermore, the goal of a repair policy is not necessarily aimed at fixing the underlying vulnerability. Instead, it should supply a configurable response that augments the wide variety of protection mechanisms available to serve as a short-term "band-aid." Like SELinux policies, default repair policies can be written and distributed by application vendors, stakeholders, or competitors. The key insight here is that these roles have a distinct advantage over developers: at the time that vendors, system administrators, and other stakeholders need to write repair policy, they have access to the completed system. Developers, on the other hand, are still in the middle of constructing the software and should not attempt to simultaneously construct repair policy.

"Fixing" the source or waiting for a patch is beyond the scope of the problem we are considering. Protection mechanisms already exist that detect large classes of attacks. We focus on the complementary issue of what should be done when any attack is detected, as

waiting an indeterminate amount of time for a vendor to generate and test a patch may not be acceptable. A policy should apply to TPs containing both known and unknown vulnerabilities.

It is important to note that the process of self–healing does not necessarily hide or mask errors. If it did, security is decreased because a broken system is being run without the knowledge of the administrator. Self–healing does not preclude the system from reporting errors, alerts, or policy violations so that a human effort to address the source of the vulnerability parallels the automatic process, and we note that CW has strict audit requirements.

One difficulty with automatic supervision is that attacks and faults can be relatively rare events, and inserting IRM checks throughout the application's entire execution might needlessly impact the normal operation of the software. As we discuss in Chapter 4, STEMv2 has flexible coverage policies.

5.5.3 Future Work

Our future work on Ripple centers on improving the power and ease of use of the language. It became clear during our testing that Ripple must deal with dynamically allocated data structures. We intend to provide a more robust mapping between memory layout and source-level variables. Cutting across layers of abstraction like this requires augmenting the existing mapping mechanism with a more extensive type system and the ability to handle variables that do not reside at fixed addresses.

We intend to explore the use of virtual proxies (see Chapter 9) as a construct in the Ripple language. Ripple's interaction with I/O is limited to the memory locations that are sources or sinks and the `stdout` channel. We also intend to explore the addition of formal logic to Ripple's virtual machine so that it can reason about the constraints on the data involved in a transaction to learn the best response over time. Finally, the information that a particular set of variables have been corrupted raises the possibility of notifying other hosts and application instances to proactively invoke repair procedures in order to protect against a widespread attack [LSK06a, CCCR05, TLH+07]. This sort of detection can help a system automatically tunes an organization's security posture.

Chapter 6

Maintaining an External Integrity Posture

Maintaining an external integrity posture is a critical part of a software self–defense system, and doing so complements the actions necessary to protect an internal integrity posture. The intuition is compelling: if malformed or malicious input never reaches the software application, neither the original fault nor the self–healing instrumentation need be exercised to protect the *internal* integrity posture of the system. The key research question is how to keep the input filtering mechanism updated with the details of new attacks. In turn, we must accomplish three tasks: reliably detect new attacks, extract forensic information from the sensors, and create filters that capture the essence of those attacks. Doing so can prevent the protected application (or cooperating peers) from consuming similar attack messages in the future.

Automatic generation of exploit filters, despite recent work on showing the infeasibility of detecting polymorphic threats [SLS+07] (the impact of this work is considered in Section 6.5), remains an attractive method of protection for a few reasons.

1. Filtering is relatively easy to implement. It is usually accomplished with traditional, mature, and thoroughly researched state machine, parsing, and string matching algorithms.

2. Filtering and string matching have fairly well–understood runtime costs. Hardware,

parallel, or optimized implementations make the prospect of matching large amounts of traffic or other input data a reasonable enterprise, at least at the network edge or when scanning local large data sets.

3. The cost of filtering can be amortized over a network of hosts or collection of applications. Filtering at well–defined locations in the network can benefit a large number of hosts while only slightly increasing management burden.

4. Filtering is non-invasive. Filtering stops malicious I/O before it affects the control flow of the target application. Filtering machinery can be physically and virtually separated from the mainline execution of an individual program or host. Filtering code need not clutter the source or instruction stream of a process[1]. A separate filtering component presents another "attack surface" for adversaries[2]. While this additional system component may increase management complexity, on the whole, we believe that complexity is balanced by the benefits of defense-in-depth and simpler application codebases.

In this chapter, we consider the design and implementation of FLIPS (Feedback Learning Intrusion Prevention System), our signature generation and exploit filter enforcement mechanism. We focus on protecting an HTTP server for concreteness and because such software is a common attack target. The goal of the system is to provide a modular application-level firewall with the ability to automatically learn and drop confirmed new or "0-day" attacks. Only attacks that both inject and execute code are confirmed as malicious and supplied to the anomaly sensor and filter generator.

6.1 Automatic Signature Generation

In large part, signatures of viruses and other malware are currently produced by manual inspection of the malware code. Involving humans in the response loop dramatically

[1]If filtering was interwoven with these artifacts, then debugging and maintenance and the complexity of the software increase, which usually has a detrimental effect on security.

[2]This additional attack surface is both good and bad: it presents a potential weak point for an attacker, but it also presents an additional obstacle to surmount.

lengthens response time and does nothing to stop the initial infection. In addition, deployed signatures and IDS rules do nothing to guard against new threats. Traditional network defense systems (*e.g.,* Intrusion Detection Systems (IDS) and firewalls) have shortcomings that make it difficult for them to identify and characterize new attacks and respond intelligently to them. Since an IDS only passively classifies information, it can enable but not enact an automated response or defense. Both signature–based (*i.e.,* misuse–based) and anomaly–based approaches to classification and detection merely warn that an attack *may* have occurred. Attack prevention is a task often left to a firewall, and it is usually accomplished by string–matching signatures of known malicious content or dropping packets according to site policy. Of course, successfully blocking the correct traffic requires a flexible and well defined policy. In addition, encrypted and tunneled network traffic poses problems for both firewalls and IDS. To compound these problems, since neither network IDS or firewalls know for certain how an end host may process a packet, they may make an incorrect decision [HPK01].

These obstacles motivate the argument for placing protection mechanisms closer to the end host (*e.g.,* distributed firewalls [IKBS00]). This approach to system security can benefit not only enterprise networks, but the private machines of end users as well. The principle of "defense-in-depth" suggests that traditional perimeter defenses be augmented with host–based protection mechanisms. This chapter presents an example of one such system.

6.2 FLIPS

FLIPS incorporates three major components: a content anomaly classifier, a filtering scheme, and the two versions of our supervision framework, STEM. FLIPS uses STEM to capture the injected code and feed it back to the filter generation mechanism. The filter can discard input that is anomalous *or* matches known malicious input, effectively protecting the application from additional instances of an attack. FLIPS does not require authentication of a known user base and can be deployed transparently to clients and with minimal impact on servers. We can use the metrics proposed by Smirnov and Chiueh [SC05] to classify FLIPS: it detects attacks, identifies the attack vector, and provides an automatic repair mechanism.

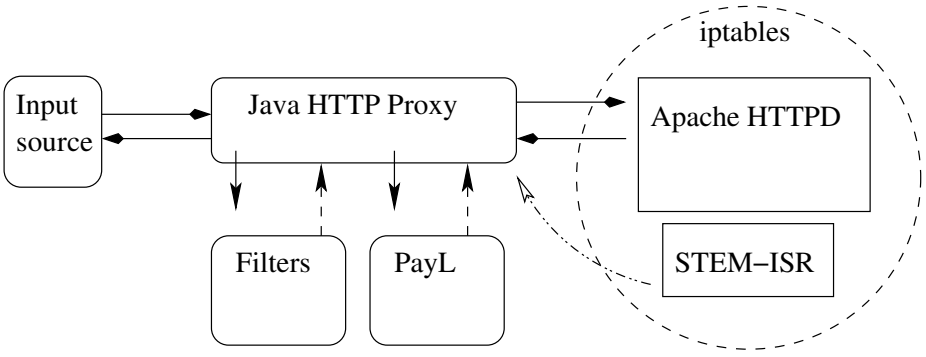


Figure 6.1: *FLIPS Prototype.* We constructed an HTTP proxy to protect HTTP servers (in this example, Apache) from malicious requests. The proxy invokes a chain of three filtering mechanisms and PAYL to decide whether or not to drop each HTTP request. Apache is also protected by a packet filter that denies requests from hosts other than the proxy.

6.2.1 Hybrid Detection

In general, detection systems that rely solely on signatures cannot enable a defense against previously unseen attacks. On the other hand, anomaly-based classifiers can recognize new behavior, but are often unable to distinguish between previously unseen "good" behavior and previously unseen "bad" behavior. This blind spot usually results in a high false positive rate [WCS05] and requires that these classifiers be extensively trained[3].

The architecture of FLIPS contains a proxy that houses a hybrid anomaly classifier and filtering mechanism. This proxy is complemented by STEM acting as a supervision framework. We adopted this architecture because, as Chapter 2 explains, anomaly detection methods suffer from a high false positive rate. Due to both unclean training data sets as well as the problem of "model drift" (where the normal behavior of a system gradually diverges from the trained model), anomaly sensors require an additional source of information that can confirm or reject their initial classification. Pietraszek [Pie04] presents a method that uses supervised machine learning to tune an alert classification system based on observations of a human expert. In contrast, FLIPS *automatically* confirms or rejects the anomaly sen-

[3]Cretu *et al.* [CSSK07, CSL+08] consider how to clean training data sets using feedback from systems like FLIPS.

sor's conjecture or hypothesis based on feedback from STEM. We use an anomaly detector based on PAYL [WS04, WCS05], but other classifiers can be used [KTK02, WPS06].

STEM allows us to capture injected code and correlate it with input that has been classified as anomalous. In the case of STEMv1's use of instruction set randomization (ISR), Barrantes *et al.* [BAF+03] show that code injection attacks against protected binaries fail within a few bytes (two or three instructions) of control flow switching to the injected code. Therefore, the code pointed to by the instruction pointer at the time the program halts is (with a high probability) malicious code. We can extract this code and send it to our filter to create a new signature and update our classifier's model. STEMv2's detection mechanisms can extract the same or similar forensic memory information.

6.2.2 Proxying: Classification and Filtering

We insert a proxy between I/O sources & sinks and the application we protect with FLIPS. Its purpose is to provide an environment where we can classify and filter anomalous, malformed, or malicious requests or messages before that I/O affects the application. This multi-component proxy is the key guardian of our external integrity posture, and it reflects the dichotomy of our classification schemes.

A chain of signature filters score and drop a request if it matches known malicious data. A chain of anomaly classifiers can score and drop the input if it is outside the normal model. The default policy for FLIPS is to only drop requests that match a signature filter. Requests that the anomaly classifier deems suspicious are copied to a cache and forwarded on to the application. We adopt this stance to avoid dropping requests that the anomaly component mislabels (false positives). The current implementation only drops requests that have been confirmed to be malicious to the protected application as well as requests that appear similar to such inputs.

The HTTP proxy is a simple HTTP server that spawns a new thread instance for each incoming request. During the service routine, the proxy invokes a chain of Filter objects on the HTTP request. Our default filter implementation maintains three signature–based filters and a Classifier object. PAYL implements the Classifier interface to provide an anomaly score for each HTTP request. When the proxy starts, it creates an instance of

PAYL and provides a sample traffic file for PAYL to train on.

The core of the filter implementation is split between two subcomponents. The *checkRequest()* method performs the primary filtering and classification work. It maintains four data structures to support filtering. The first is a list of "suspicious" input requests (as determined by PAYL). This list is a cache that provides the feedback mechanism a starting point for matching confirmed malicious input. Note that this list is not used to drop requests. The remaining data collections form a three–level filtering scheme that trades off complexity and cost with a more aggressive filtering posture. These lists are not populated by PAYL, but rather by the feedback mechanism. The first level of filtering is direct match. This filter is the least expensive, but it is the least likely to block malicious requests that are even slightly polymorphic. The second filter is a reverse lookup filter that stores requests by the score they receive from PAYL (similar scores may indicate clustered related anomalous requests). Finally, a longest common substring filter provides a fairly effective means of catching malicious requests, even though the runtime behavior is more expensive than the other two $O(1)$ matching schemes. An example of an exploit rejected by the direct match filter is shown in Figure 6.2. The score reported after the REJECT decision is PAYL's score for the request.

```
...
[HTTPServerMain] Mon Aug 13 14:14:21 EDT 2007: \
[RequestHandler-8] Found (lafilter.fproxy.http.httpd.ProxyEngine) \
 to service request
[Filter] Mon Aug 13 14:14:21 EDT 2007: trying to filter entry \
  [GET /exploitSA?...Jgnnmumpnf... HTTP/1.0]
[Filter-1] Mon Aug 13 14:14:21 EDT 2007: Direct match REJECT. \
  score = 0.7552812376187079
...
```

Figure 6.2: *Exploit Filtering.* The direct match filter has rejected this particular request (we provide an obfuscated extract of the real payload). We note that PAYL also deems this particular request highly anomalous (PAYL scores from 0..1, and values over .5 can be considered anomalous). Nevertheless, the rejection is based solely on the signature matching, not PAYL. We can treat this decision as an affirmation of PAYL's classification to improve its efficacy.

6.2.3 Feedback Learning

The feedback mechanism in the proxy is provided by a background thread listening for notifications from STEM about malicious binary code. This thread simply reads in a sequence of bytes and checks if they match previously seen "suspicious" input (as classified by PAYL). If not, then the thread widens its scope to include a cache of all previously seen requests. Matching is done using the longest common substring algorithm. If a match is found, then that request is used in the aforementioned filtering data structures. If not, then a new request is created and inserted into the filters based on the malicious byte sequence. An example of generating a signature based on feedback from STEM is shown in Figure 6.3.

```
...
[Learner] Mon Aug 13 14:13:55 EDT 2007: searching SUSPICIOUS_LIST(0) \
  [GET /exploitSA?...Jgnnmumpnf... HTTP/1.0]
[Learner] Mon Aug 13 14:13:55 EDT 2007: found match (7%) in SUSPICIOUS_LIST
[Learner] Mon Aug 13 14:13:55 EDT 2007: no matches in SEEN_LIST
[Learner] Mon Aug 13 14:13:55 EDT 2007: inserted new entry \
  [GET /exploitSA?...Jgnnmumpnf... HTTP/1.0] \
  of length 126 bytes into DIRECT_MATCH
...
```

Figure 6.3: *Exploit Signature Generation*

The communication protocol between FLIPS and STEM is fairly straightforward. FLIPS expects a range of raw byte values and then performs the heavy lifting of matching against requests. To complement this work, STEM performs some minor memory forensics to extract bytes involved in the attack. An example of STEM's operation during the feedback procedure is shown in Figure 6.4. STEM essentially captures a section of memory whose size is based on the value of the `frame_radius` parameter shown in Figure 4.2. The `hostname` and `port` configuration parameters from Figure 4.2 indicate the communication endpoint where FLIPS's feedback thread is listening for messages. STEM captures two distinct types of information: (1) the current state of the stack (useful for capturing malcode injected during the exercise of a stack–based buffer overflow) and (2) the memory locations surrounding a small history of the instruction pointer. The latter type of information is especially useful for STEMv1's use of ISR; ISR may have failed after consuming a few more "illegal" instruc-

tions after the first injected instruction due to the derandomization process. In this case, keeping track of the history of the instruction pointer (the `%eip` register on `x86`) helps us identify attack code.

```
[STEM]: Received SIGSEGV. Next instruction @ emuEIP = 809f864
[STEM]: emuEIP history is:
        eip_history[0] = 0x809f862
        eip_history[1] = 0x809f85f
        eip_history[2] = 0x809f85a
        eip_history[3] = 0x809f858
        eip_history[4] = 0x809f856
        eip_history[5] = 0x809f855
        ...
[STEM]: Extracting signature from stack frame...
mem[0xbfc071d8] 32(%ebp) = 0x96bb110
...
mem[0xbfc07198] -32(%ebp) = 0x2020202
[STEM]: notify_flips @ localhost:3780
[STEM]: message is 72 bytes
[STEM]: message (grouped in words) = [0x96bb0a0 0x96bb110 0x0 \
 0xbfc071e8 0x96bb11a 0x96bb120 0x96bb110 0x96ba360 0x809fecc \
 0xbffd12fd 0xfdfd64fd 0xfdfdfd47 0xfd020202 0x17bfd02
 0x2025bfd 0xfdfd2efd 0xfdfdfd59 0x2020202 ]
FLIPS acknowledges receipt of 72 bytes.
[STEM]: return code for notify_flips() = NOTIFICATION_OK
```

Figure 6.4: *STEM Providing Feedback to FLIPS*

6.2.4 Limitations

While the design of FLIPS is quite flexible, the nature of host-based protection and our choices for a prototype implementation impose several limitations. First, host-based protection mechanisms are thought to be difficult to manage because of the potential scale of large deployments. Outside the enterprise environment, home users are unlikely to have the technical skill to monitor and patch a complicated system. We purposefully designed FLIPS to require little management beyond installation and initial training, and STEMv2 can operate on unmodified, pre-existing binaries. One task that should be performed during system installation is the addition of a firewall rule that redirects traffic aimed at the protected

application to the proxy and only allows the proxy to contact the protected application.

Second, the performance of such a system is an important consideration in deployment. We show in Chapter 8 that the benefit of automatic protection and repair (as well as generation of zero-day signatures) is worth the performance impact of the system. If the cost is still too high for normal operation, the system can be used as a sensor to protect a community of applications.

Third, the proxy should be as simple as possible to promote confidence in its codebase that it is not susceptible to the same exploits as the protected application. We implement our proxy in Java, a type-safe language that is not vulnerable to the same set of binary code injection attacks as a C program. The proxy includes PAYL (400 lines of code) and a simple HTTP proxy that incorporates the exploit filter generation and enforcement (about 5K lines of code).

6.3 Filter Generation, Enforcement, and Efficacy

Preserving the external integrity posture of a software system entails some runtime supervision cost (*e.g.,* the use of STEMv1 or STEMv2) and some network processing cost. This section characterizes the latter cost.

Inserting a filter on the communications path of an application raises concerns because of the anticipated performance impact of the detection algorithms and the validity of the decision that the detection component reaches. In this section, our primary aim is to show that the combined benefit of automatic repair and exploit signature generation is worth the price of even a fairly unoptimized proxy implementation. Our evaluation has two major aims: (1) establish the performance impact of filter generation and filter enforcement and (2) show that the system operates end-to-end and can block variations of an attack.

The first aim is accomplished by measuring the additional time the proxy adds to the overall processing with two different HTTP traces. We accomplish the second aim by performing an end-to-end test showing how quickly the system can detect an attack, register the attack bytes with the filters, create the appropriate filter rules, and drop the next instance of the attack. We send a request stream consisting of the same attack at the proxy

and measure the time (in both number of "slipped" attacks[4] and real time) it takes the proxy to filter the next instance of the attack. We evaluate the performance overhead of the filter enforcement and filter generator by using a simple client to issue requests to a "production" Web server and measure the change in processing time as each proxy subcomponent is introduced. Table 6.1 describes these results.

6.3.1 Experimental Setup

The experimental setup includes an instance of Apache 2.0.52 as the production Web server with one modification to the basic configuration file: the "KeepAlive" attribute was set to "Off" to avoid servicing multiple requests with one thread. Then, an `awk` script reconstructed HTTP requests from a dump of Web traffic and passed the request over the `netcat` utility to either the production server or the proxy (as befitting the stage of performance testing; in a real deployment, all traffic goes through the proxy). The proxy was written in Java, compiled with the Sun JDK 1.5.0 for Linux, and run in the Sun JVM 1.5.0 for Linux. The proxy was executed on a dual Xeon 2.0GHz with 1GB of RAM running Fedora Core 3, kernel 2.6.10-1.770_FC3smp. The production server platform runs Fedora Core 3, kernel 2.6.10-1.770_FC3smp on a dual Xeon 2.8 GHz processor with 1GB of RAM. The proxy server and the production server were connected via a Gigabit Ethernet switch. The servers were reset between tests. Each test was run for 10 trials.

The platform that FLIPS was deployed on (Fedora Core 3) employs address space randomization. We turned this mechanism off by changing the value in */proc/sys/kernel/exec-shield-randomize* to zero. In addition, we marked the `httpd` binary as needing an executable stack via the `execstack` utility. We take these actions in order to enable the attacks to be detected with our ISR mechanism rather than be prevented with the OS's ASR and non-executable stack mechanisms.

6.3.2 Proxy Performance

We discovered the performance impact of our unoptimized, Java-based proxy on the time it took to service two different traffic traces. We note that the cost of training PAYL is

[4]Think of "slipped attacks" as the amount of oil that escapes while trying to cap a burning oil well.

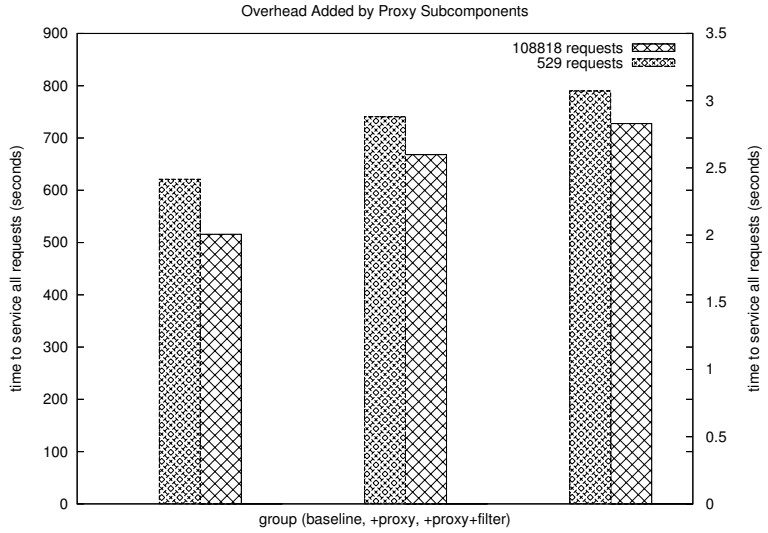


Figure 6.5: *Performance Impact of FLIPS Proxy Subcomponents*. A demonstration of how the proxy affects baseline performance for two different traffic traces. Note that the smaller trace (529 requests) is measured on the vertical axis on the right side of the graph. This graph shows the increase in average time to service requests when the proxy is inserted between the client and the HTTP server with and without filtering activated.

incurred once at system startup and takes about 5 seconds for a 5MB file of roughly 109000 HTTP requests. Our performance results are displayed in Table 6.1 and graphically in Figure 6.5. Note that our experimental setup is not designed to stress-test Apache or the proxy, but rather to elucidate the relative overhead that the proxy and the filters add. Baseline performance is roughly 210 requests per second. Adding the proxy degrades this throughput to roughly 170 requests per second. Finally, adding the filter reduces it to around 160 requests per second. This overhead comes from a few sources.

First, the basic cost of serving as a proxy, including reading data from the network and parsing it for sanity occurs for each request. Second, we use an interpreted language (Java) to implement the proxy and anomaly sensor. Some of our implementation choices also impose a bottleneck; the proxy is multi-threaded but synchronized at one FilterManager object. Additionally, the anomaly sensor is invoked on each request.

Table 6.1: *Performance Impact of FLIPS Proxy Subcomponents.* Baseline performance is compared to adding FLIPS's HTTP proxy alone and FLIPS's HTTP proxy with filtering and classification turned on.

	# of Requests	**Mean Service Time (s)**	**Std. Dev.**
Baseline	529	2.42	0.007
Baseline	108818	516	65.7
+Proxy	529	2.88	0.119
+Proxy	108818	668	9.68
+Proxy, +Filter	529	3.07	0.128
+Proxy, +Filter	108818	727	21.15

6.3.3 Filter Efficacy

To demonstrate the operation of the system, we inserted three synthetic code injection vulnerabilities into Apache. The basic vulnerability was a simple stack-based overflow of a local fixed-size buffer, as well as other variations of stack overwrites. The function was protected with STEMv1, and we observed how long it took FLIPS to stop the attack and deploy a filter against further instances.

To test the end-to-end functionality, we directed two streams of attack instances against Apache through our proxy. We first sent a stream of 67 identical attack instances and then followed this with 22 more attacks that included slight variations of the original attack. Once FLIPS has had feedback from STEM, it will block all future identical attack instances.

In the first attack stream, FLIPS successfully blocked 61 of the 67 attack instances. It let the first six instances through before STEM had enough time to feedback to FLIPS. It took less than one second for FLIPS to start blocking the attacks. After that, each subsequent identical attack instance was blocked by the direct match filter.

The second attack stream contained 22 variations of the original. The LCS filter (with a threshold of 60%) successfully blocked twenty of these without any false positives. Most of the blocked attacks had an LCS of 80% or more. Obviously, attacks that are extremely different will not be caught by the LCS filter, but if they cause STEM to signal FLIPS

about them, they will then be blocked on their own merits.

6.4 Expanding Exploit Filter Coverage

Matching static strands of string content against arbitrary amounts of raw network messages does not provide a satisfactory solution, as attack messages may vary slightly in composition due to either the attacker's explicit attempts to evade the filtering mechanism or the vagaries of network operation. An attacker could also attempt to trick the system into rejecting benign content by using an allergy attack [CM06]. One way in which FLIPS attempts to alleviate or address these concerns is by using the longest common substring algorithm to match portions of attack messages. While this approach is a good start, another way to strengthen the filtering mechanism is to become aware of the protocol format. By fixing parts of the exploit instances to a relative offset within a specific protocol field (rather than absolute offsets in an opaque, single exploit sample), we can generalize the exploit filter and learn which parts of the exploit are flexible in terms of content. The work we relate in this section is similar to that performed by Cui *et al.* [CPWL07].

A particular configuration of parsing states can be thought of as a vulnerability description, or at least a more flexible exploit description. In short, we can use protocol knowledge to group stages of program behavior that correspond to parsing and input handling. Matching the parts of a sample exploit with specific fields in the protocol grammar can make the filters more resistant to minor perturbations in attack messages. It is less likely that changes like single character insertions, deletions, or modifications will make the exploit completely unrecognizable to the filter. If the filter is protocol-aware, then it is more robust than simple string matching because the protocol context indicates when content is in the same context as a malicious request.

Furthermore, we can explore the space of the exploit by transforming the exploit according to the grammar rules and vetting the modified exploit with a reliable sensor to discover irrelevant portions of the exploit message. Such a procedure is, in fact, a local search around the space of the sample exploit. Performing this search helps generalize the exploit filter by removing overly specific constraints. Alternative techniques include the Packet Vaccine

approach [WLX+06]. The main idea is to capture an exploit instance with STEM or some other reliable detection mechanism and then parse it with a protocol parsing engine. We use a detection mechanism similar to that of Vigilante [CCCR05], and our protocol parsing engine is provided by an extension of the GAPA framework [BBW+07].

The core of the system is based on a feedback loop involving the *Detector* (which we treat as an oracle), the protocol parsing engine, a *Refinery*, a *FilterGeneralizer*, and a *VirtualPacketModifier*. We treat input to the system as a sequence of *Virtual Packets*. A VirtualPacket may be an IP or TCP packet, a higher–level protocol message, or a chunk of file data. The Detector initially detects an exploit due to the use of the data represented by the VirtualPacket. It outputs the VirtualPacket and a set of *WatchBytes*: byte positions (and values) for the feedback system to monitor.

The Refinery accepts as input the VirtualPacket stream and the set of WatchBytes. The Refinery uses the services of the protocol parsing engine to map parts of the exploit (as identified by the WatchBytes) to a particular protocol state. The Refinery thus *refines* the alert information. When the data parser consumes a field, the Refinery checks if the consumed range includes a position identified in a WatchByte. If so, the Refinery records the terminal name, field length, and relative offset within the field that the WatchByte occurs. The Refinery also records the protocol state and the parsing state (the value of sibling nodes, current stack, along with the values of previous terminals and their then–current parsing stack states). The Refinery then uses this information to create a very specific exploit filter. The parsing engine can use this filter to block that particular exploit message by checking that the parsing state matches the conditions that the Refinery has identified. An example of such a filter is shown in Figure 6.6.

Once this filter *generation* process has completed, we can turn to the process of filter *generalization*. The first filter is specific to the particular exploit instance we captured. The process of generalizing the filter involves discarding conditions in the filter that needlessly constrain the format and content of the exploit. For example, in the WMF exploit, the malcode payload does not matter for the actual exercise of the vulnerability, so a condition on this field is not needed, and an attacker could frustrate us by simply including slightly different payload code).

Filter generalization is accomplished via a cycle of modifying the original VirtualPacket with a series of transformations according to the protocol grammar. We generate a list of terminal symbols to change and then pass this list of *ChangeTerminal*s to the *VirtualPacketModifier*. The VirtualPacketModifier parses the VirtualPacket in a fashion similar to the Refinery, except that it checks if a terminal symbol is a registered ChangeTerminal. If so, it checks if the parsing state stack of the ChangeTerminal matches the current stack. If so, it alters the value of the terminal (optionally discarding it if allowed by the protocol) and replaces it in the VirtualPacket stream. The VirtualPacketModifier regenerates the VirtualPacket and passes it to the Detector. If the VirtualPacket still trips the Detector, the altered terminals may not have been critical for the exploit to succeed. We can discard conditions related to these terminals. If the VirtualPacket does not trip the Detector, then the field was important to the exploit.

In our experiments, we generated filters for the WMF vulnerability, the SQL Slammer worm, and the MS Blaster exploit. The process of generalization does not introduce false positives because modified VirtualPackets are vetted by the Detector. We note that this approach to exploit signature generalization does not produce a vulnerability signature — it produces a more general exploit signature. The process requires protocol knowledge, and this knowledge may not match the code's logic. In essence, the protocol parser, like many network components, lacks complete knowledge of the end–host state.

6.5 Countering Polymorphism

Polymorphic malcode remains a troubling threat. The ability for malcode to automatically transform into semantically equivalent variants frustrates attempts to rapidly construct a single, simple, easily verifiable representation. Conventional wisdom has held that attackers retain a significant advantage by using polymorphic tactics to disguise their shellcode. To the best of our knowledge, Song *et al.* [SLS+07] present the first quantitative analysis of this advantage by providing measures of the variability and propagation strength of shellcode polymorphism. This analysis provides empirical evidence to support the folk wisdom about the perceived and actual difficulty of the polymorphic shellcode problem.

The authors focus on the nature of shellcode *decoding routines*: one of the most constrained sections of a malcode sample. Since this section of a malcode sample or exploit instance must contain executable code, it cannot easily be disguised via polymorphism (unlike most other parts of a malcode sample, except, perhaps, the higher order bits of the return address section).

Song *et al.* employ directed local search of the space of n-byte decoders (for reasonable values of n) using genetic algorithms. The empirical evidence they gather from this procedure indicates that modeling the *class* of self–modifying code is likely intractable (it is too greatly spread and varied) by known methods, including both statistical constructs and string signatures.

The work of Song *et al.* shows that almost perfect polymorphism is, if not trivial, relatively easy to achieve. This work suggests that one reason current threats have not adopted *more* polymorphism is that current defenses do not take advantage of even simple automated exploit signature generation schemes like that presented in this chapter. Should such signature generation schemes experience widespread adoption, attackers will certainly shift to using different forms of the "near-perfect" hybrid polymorphism engine Song built. Simple static string signatures — even if they are automatically generated — are doomed because the attacker can arbitrarily increase the size of the signature database, causing the filtering process to suffer extraordinary slowdown[5]. In short, the state of the art needs to move from input content filters to "behavior" filters, where "behavior" is a spectrum ranging from string content signatures to a program interpreting the behavior of the malcode.

For the immediate future, filtering input for exploit instances remains viable, but we must eventually adopt behavior filters. A behavior filter essentially interprets a relation between k exploit instances and m internal behaviors. This relation maps, groups, or classifies inputs by how they affect the internal integrity posture of a software application. We intend to investigate methods for doing so in future work, and we note that a number of other researchers are focused on producing vulnerability signatures [CCZ+07, CPWL07, NBS06] to filter various aspects of "compromised" behavior.

[5]Perhaps massively multi-core and many-core hardware can be used to support such large-scale filtering, but this use seems like a waste of resources in the long run.

6.6 Summary

FLIPS contributes a comprehensive and realistic system that uses information confirming an attack to generate exploit filters. STEM identifies exploit code, automatically recovers from the attack (as per our self–healing and repair policies), and forwards the exploit information to FLIPS. We have also described one method of making the exploit signatures less sensitive to minor perturbations of attack content. We could exchange signatures with other instances of FLIPS (or similar software) via a centralized trusted third party, a peer-to-peer network, or other distribution mechanism. Such information exchange [CM02, KMLC05] can potentially inoculate the network against a zero-day worm attack [AGI+03, KK04, Sto04] or other threats. While our examples demonstrate an implementation of FLIPS that protects an HTTP server, FLIPS's mechanisms are broadly applicable to other software systems. More detail on our experimental validation of FLIPS is given in Chapter 8.

```
handler WMFScanner(WMF_File)
{
   bool vp_malicious0 = false;  bool vp_malicious1 = false;
   bool vp_malicious2 = false;  bool vp_malicious3 = false;
   bool vp_malicious4 = false;

   @WMF_File_A ->{vp_malicious0 = true;}
   @RegularMetaFile ->{vp_malicious1 = true;}
   @RecordList_A ->{vp_malicious2 = true;}
   @Record ->{vp_malicious3 = true;}
   @RecordBody ->{vp_malicious4 = true;}
   @EscapeBody ->{
      print("@[EscapeBody]\n");
      if(vp_malicious0 && vp_malicious1 && vp_malicious2 &&
         vp_malicious3 && vp_malicious4 && true)
      {
         if(escape_number==9 &&
            num_escape_data_bytes==88 &&
            escape_executable_body==...)
         {
            print("***alert***\n");
            return S_ReadFile;
         }
      }
   }
   return S_ReadFile;
};
```

Figure 6.6: *Dynamically Generated Example Exploit Filter.* Such filters can be thought of as a rule for a stateful firewall. The filter is event-driven; it keeps track of the parsing state of data messages. If the parsing stack state is equivalent to one in which the application has already been exploited, then we test the contents of the terminal symbols (the fields) of the message. If this data matches, then the message is an exploit. In our experiments, the "malcode" (which we omit from this figure) in the message body simply pops up a Windows MessageBox. This filter has not yet been generalized. The purpose of filter generalization is to discard conditions on the input that are too restrictive. In the WMF case, the contents of `escape_executable_body` are unimportant, as they represent the payload of the exploit, not the vulnerability condition.

Chapter 7

Behavior Profiling

Current embryonic attempts at software self–healing produce mechanisms that are often oblivious to the semantics of the code they supervise. We believe that, in order to help inform runtime repair strategies, such systems require a more detailed analysis of dynamic application behavior. We describe how to profile an application by analyzing all function calls (including library and system) made by a process. We create predictability profiles of the return values of those function calls. Self–healing mechanisms that rely on a transactional approach to repair (that is, rolling back execution to a known safe point in control flow or slicing off the current function sequence) can benefit from these return value predictability profiles. We focus on the use of function return values as the main feature of these profiles because they can help drive control flow decisions after a self–healing repair.

Profiles built for the applications we tested can predict behavior with 97% accuracy given a context window of 15 functions. We also survey the distribution of return values for real software and present a novel way of visualizing both the macro and micro structure of these distributions. Our behavior profiling techniques help demonstrate the feasibility of combining binary–level behavior profiling with self–healing repairs.

7.1 Observing Program Behavior

A popular approach to observing program behavior utilizes anomaly detection on a profile derived from system call sequences [MRVK07, CC02, SF00, FKF+03, GRS04]. Relatively

little attention has been paid to the question of building profiles — in a non-invasive fashion — at a level of detail that includes the application's *internal* behavior. In contrast, typical system call profiling techniques characterize the application's interaction with the operating system. Because these approaches treat the application as a black box, they are generally susceptible[1] to mimicry attacks [WS02]. Furthermore, the increasing sophistication of software attacks [CXS+05] calls into question the ability to protect an application while remaining at this level of abstraction (*i.e.,* system call interface).

Behavior profiling has been used to create policies for detection [Pro03, LcC04]. In contrast, we suggest using this information to help automatically generate templates for repair policies [LSCK07]. In addition, this information can drive the selection of "rescue points" for the ASSURE system [SLKN07]. One goal of this chapter is to provide systems like STEM and ASSURE with a profiling mechanism. STEM collects aspects of data and control flow to learn an application's behavior profile. STEM can leverage the information in the profile to detect misbehavior (*i.e.,* deviation from the profile).

7.2 Profile Structure

In profiling mode, STEM dynamically analyzes all function calls made by the process, including regular functions and library calls as well as system calls. Previous work typically examines only system calls or is driven by static analysis. STEM collects a feature set that includes a mixture of parent functions and previous sibling functions. STEM generates a record of the observed return values for various invocations of each function.

A behavior profile is a graph of execution history records. Each record contains four data items: an identifier, a return value, a set of argument values, and a context. Each function name serves as an identifier (although an address or callsite is also used occasionally). A mixture of parents and previous siblings compose the context. The argument and return values correspond to the argument values at the time that function instance begins and ends, respectively. STEM uses a pair of analysis functions (inserted at the start and end of

[1] Gao *et al.* [GRS05] discuss a measure of behavioral distance where sequences of system calls across heterogeneous hosts are correlated to help avoid mimicry attacks.

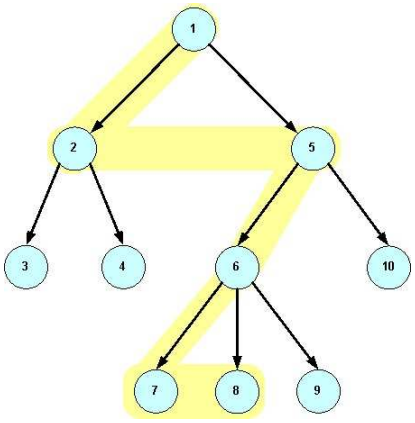


Figure 7.1: *Example of Computing Execution Window Context.* Starting from function 8, we traverse the graph beginning from the previously executed siblings up to the parent. We recursively repeat this algorithm for the parent until we either reach the window width or the root. In this example, the window contains functions 7, 6, 5, 2, 1. Systems that examine the call stack would only consider 6, 5, and 1 at this point.

each routine) to collect the argument values, the function name, the return value, and the function context.

Each record in the profile helps to identify an instance of a function. The feature set "unflattens" the function namespace of an application. For example, `printf()` appears many times with many different contexts and return values, making it hard to characterize. Considering every occurrence of `printf()` to be the same instance reduces our ability to make predictions about its behavior. On the other hand, considering all occurrences of `printf()` to be separate instances combinatorially increases the space of possible behaviors and similarly reduces our ability to make predictions about its behavior in a reasonable amount of time. Therefore, we need to construct an "execution context" for each function based on both control flow (predecessor function calls) and data flow (return & argument values). This context helps collapse *occurrences* of a function into an *instance* of a function. Figure 7.1 shows an example context window.

During training, one behavior aspect that STEM learns is which return values to predict based on execution contexts of varying window sizes. The general procedure attempts

to compute the prediction score by iteratively increasing the window size and seeing if additional information is revealed by considering the extra context.

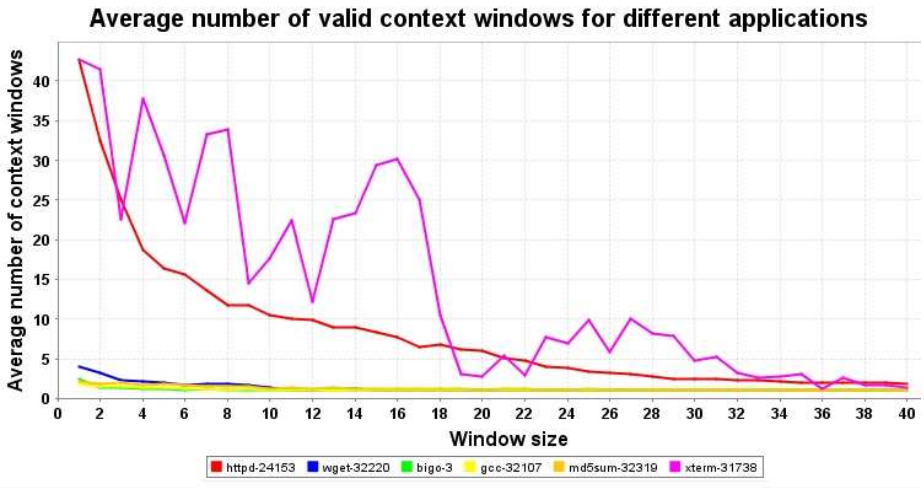


Figure 7.2: *Drop in Average Valid Window Context.* This graph shows that the amount of unique execution contexts we need to store to detect changes in control flow decreases as window size increases. `xterm` is a special case because it executes a number of other applications. If we consider the ratio of valid context windows to all possible permutations of functions, then we would see an even sharper decrease.

We adopt a mixture of parents and siblings to define a context for two reasons. First, a flat or nil context contains very little information to base a return value prediction on. Second, previous work focuses on the state of a call stack, which consists solely of parent functions. As our results in Section 7.3 demonstrate, the combination of parents and siblings is a powerful predictor of return values. The window size determines, for each function whose profile is being constructed, the number of functions preceding it in the execution trace that will be used in constructing that profile. Figure 7.2 provides insight: it shows that the amount of unique execution contexts drops as the window size increases. In contrast, if there were a large amount of valid windows, our detection ability would be diminished.

We define the return value "predictability score" as a value from zero to one. For each context window, we calculate the "individual score": the relative frequency of this particular window when compared with the rest of the windows leading to a function. The

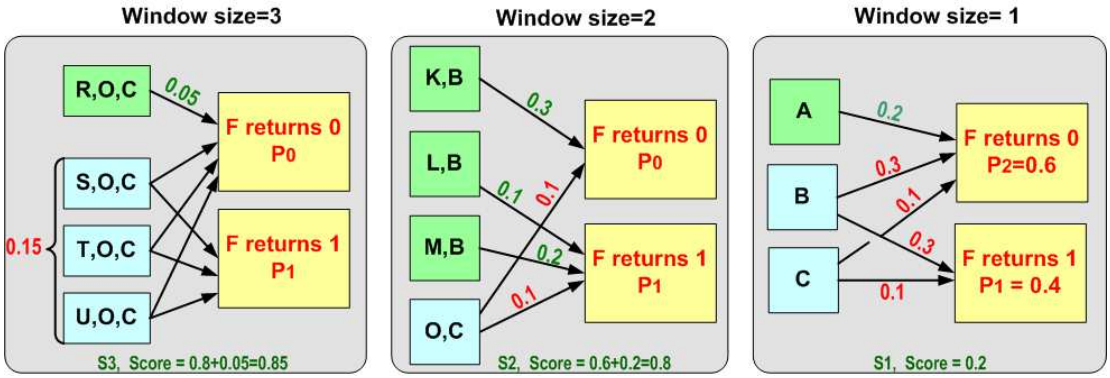


Figure 7.3: *Example of Computing Return Value Predictability (predictability score).* The figure illustrates the procedure for function F and for two return values 0 & 1 for three window sizes. The arrow labels indicate what percentage of instances for the given window will lead to the return value of F when compared with the rest of the windows. For window size 1 (S1) we have three predicate functions (A, B, and C) with only one, A, leading to a unique return value with score 0.2. This score is the relative frequency of window AF,(2) when compared with all other windows leading to F, for all return values. We add a score to the total score when a window leads to single return value of F since this situation is the only case that "predicts" a return value. We consider only the smallest windows that lead to a single value (*e.g.,* A is no longer considered for S2 and KB, LB, MB for S3) because larger windows do not add anything to our knowledge for the return value.

predictability score for a function F is the sum of the individual scores that lead to a single return value. Figure 7.3 displays an example of this procedure. We do not consider windows that contain smaller windows leading to a single return value since the information that they impart is already subsumed by the smaller execution context. For example, in Figure 7.3, we do not consider all windows with a suffix of AF (*i.e.,* $*AF$).

7.3 Evaluating Profile Generation

We start by assessing the feasibility of generating profiles that can predict the return values of functions. This section considers how to generate reliable profiles of application execution

behavior for both server programs and command line utilities. These profiles are based on the binary call graph features combined with the return values of each function instance.

We test and analyze applications that are representative of the software that runs on current server and desktop Unix environments, including: `xterm` (X.Org 6.7.0), `gcc` (GNU v3.4.4), `md5sum` (v5.2.1), `wget` (GNU v1.10.2), the `ssh` client (OpenSSH 3.9p1) and `httpd` (Apache/2.0.53). We also employ some crafted test applications to verify that both the data and the methods used to process them are correct. We include only one of these applications (`bigo`) here because it is relatively small, simple to understand, and can easily be compared against profiles obtained from the other applications. The number of unique functions for all these applications is: xterm, 2111; gcc, 294; md5sum, 239; wget, 846; ssh, 1362; httpd, 1123; and bigo, 129.

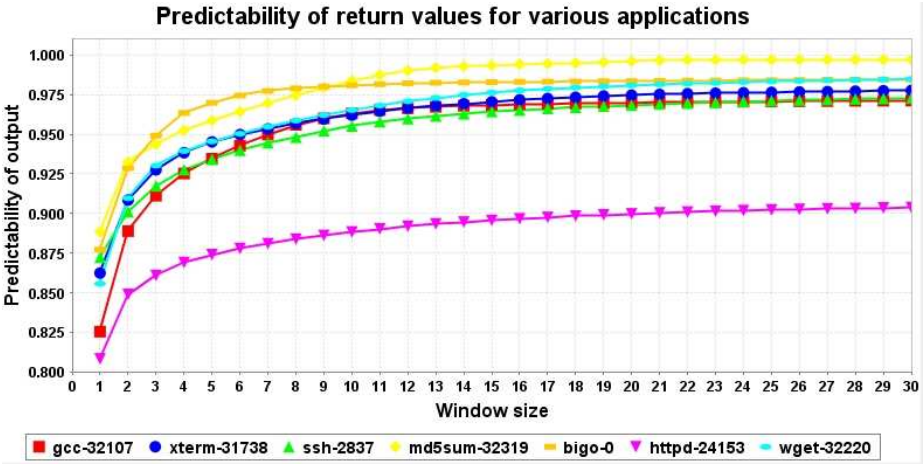
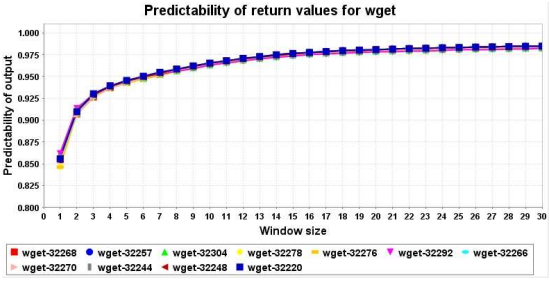
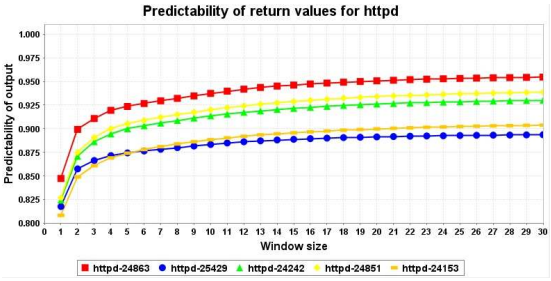


Figure 7.4: *Average Predictability of Return Values.* Return value prediction for various applications against a varying context window size. Window sizes of 15 or more achieve an average prediction of 97% or more for all applications other than `httpd` (with a rate of about 90%).

Finding a suitable repair for a function, in the context of the self–healing systems we have been discussing, entails examining the range of return values that the function produces. The notion of return value "predictability" is defined as a value from 0..1 for a specific context window size. A predictability value of 1 indicates that we can fully predict the function's return value for a given context window.



(a) *Average Predictability of Return Values for Different Runs of* `wget`. Although there are 11 runs for `wget`, each individual run is both highly predictable (>98%) and very similar to the others' behavior for different window sizes.



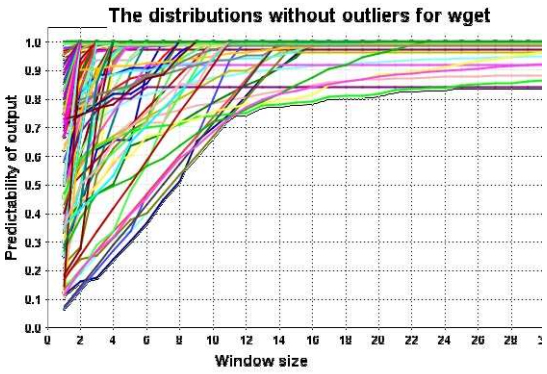
(b) *Average Predictability of Return Values for* `httpd` *and for Different Runs.* Although return value prediction remains high (>90%) for `httpd`, some variations are observable between the different runs. This phenomena is encouraging because it suggests that the profile can be specialized to an application's use at a particular site.

Figure 7.5: Return Value Predictability for both `wget` and `httpd` with Different Window Sizes

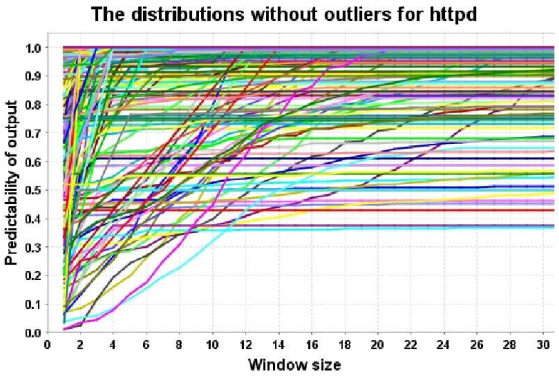
Figure 7.4 shows the average predictability for the set of examined applications. It presents a snapshot of our ability to predict the return value for various applications when we vary the context window size (*i.e.,* the history of execution). Using window sizes of more than 15 can achieve an average prediction rate of more than 97% for all applications other than `httpd`. For `httpd`, prediction rates are around 90%. This rate is mainly caused by Apache's use of a large number of custom functions that duplicate the behavior of standard library functions. Moreover, Apache is larger and more complex than the other applications and has the potential for more divergent behavior. This first data set is only a bird's eye view of an overall trend since it is based on the behavior of the average of our ability to predict function return values.

To better understand how our predictions perform, we need to more closely examine the measurements for different runs of the same application. In Figure 7.5(b) and Figure 7.5(a) we present results for different runs of `httpd` and `wget`. The `wget` utility was executed using different command line arguments and target sites. Apache was used as a daemon

with the default configuration file but exposed to a different set of requests for each of the runs. As we expected, `wget` has similar behavior between different runs: both the function call graph and the generated return values are almost identical. On the other hand, Apache has runs that appear to have small but noticeable differences. As reflected in the average plots, however, all runs still have high predictability.



(a) *Predictability of Return Values for* `wget` *Functions.* Each line represents a function and its predictability evolution as context window size increases. Most functions stabilize after a window size of ten. This graph excludes a small set (Table 7.1) of outlier functions (functions that are two standard deviations from the average).



(b) *Predictability of Return Values for* `httpd` *Functions.* Each line represents a function and its predictability evolution as context window size increases. As expected, `httpd` has more functions that diverge from the average. It also has more outliers, as shown by Table 7.1.

Figure 7.6: Per-Function Return Value Predictability for both `wget` and `httpd`

Some questions remain, including how effective our method is at predicting return values of individual functions. Also, if there are any functions that we cannot predict well, how many are there, and is there some common feature of these functions that defies prediction? Answering these questions requires measurements for individual function predictability. To visually clarify these measurements, in Figures 7.6(a) and 7.6(b), we remove functions that have a prediction of two or more standard deviations from the average. The evolution of predictability for `wget` and `httpd` is illustrated in Figure 7.6(a) and Figure 7.6(b). This evolution is consistent with Figure 7.4: most functions are predictable — and for small

context windows.

Table 7.1: *Percentage of Unpredictable (Outlier) Functions.* We illustrate the nature of the overlap in behavior profiles by examining which functions are outliers both within and across programs.

Application	**% outliers**	**% common outliers**	**% common functions**
gcc	5	53	51
md5sum	5	87	84
wget	6	62	56
xterm	6	39	17
ssh	5	35	33
httpd	10	10	28

A small percentage of the functions, however, produce return values which are highly unpredictable. This situation is completely natural: we cannot expect to predict return values that depend on runtime information such as memory addresses. Additionally, there are some functions that we *expect* to return a non-predictable value: a random number generator is a simple example. In practice, as we can deduce from our experiments (see Table 7.1), the number of such "outlier" functions is rather small in comparison to the total number of well–behaved, predictable functions.

In Table 7.1, each column represents the percentage (out of all functions in each program) of common or outlier functions. The first column presents the percentage of functions that are outliers *within* a program: that is, functions that deviate from the average profile by more than two standard deviations for all windows of size >10. For each program, there are relatively few "outlier" functions. The second column examines the percentage of common outliers: outliers that appear in two or more applications. We can see that the functions that are unpredictable are consistent and can be accounted for when creating profiles. The third column displays non-outlier functions that are common across applications. These common and predictable functions help show that some aspects of program behavior are consistent across programs.

7.4 Return Value Characteristics

While Section 7.3 shows how well we can predict return values, this section focuses on *what* return values actually form part of the execution profile of real software, and how those values are embedded throughout the model structure. We wrote a Pin tool to capture the frequency distribution of return values occurring in a selection of real software, and we created distributions of these values. These distributions provide insight into both the micro and macro structure of a return value behavior profile. Our data visualization technique scales the height and width of each cluster to simultaneously display both the intra- and inter-cluster structure. Our analysis aims to show that return values can reliably classify similar runs of a program as the same program as well as distinguish between execution models of different programs.

7.4.1 Return Value Frequency Models

Our Pin tool intercepts the execution of the monitored process to record each function's return value. The tool builds a table of return value frequencies. After the run of the program completes, we feed this data to MATLAB for a further evaluation that leads to a final return value frequency model. As a proof of concept, a model for a particular monitored process is simple, consisting of the average frequency distribution over multiple runs of the same program. Intuitively, we expect several types of clusters to emerge out of the average frequency distribution. We anticipate clusters that contain very high frequency return values, such as -1, 0, and 1 (standard error or success values as well as standard output handles). We expect a larger, more dispersed cluster that records pointer values as well as a cluster containing more "data" values such as ASCII data processed by character or string handling routines.

We examine three hypothesis dealing with the efficacy of execution behavior profiles based on return value frequency:

1. traces of the same program under the same input conditions will be correlated with their model

2. the model of all traces of one program can be distinguished from the model of all

Table 7.2: *Manhattan Distance Within and Between Models.* The diagonal (shown in italics) displays the average distance between each trace and the behavior profile derived from each trace of that program. All other entries display the distance between the execution models for each program. We omit the lower entries because the table is symmetric. Note the difference between gzip and gunzip as well as the similarity of gzip to itself.

	date	**echo**	**gzip**	**gunzip**	**md5sum**	**sha1sum**	**sort**
date	*3.03e+03*	3.72e+03	1.61e+07	1.87e+06	6.46e+04	6.47e+04	5.45e+03
echo	-	*548*	1.61e+07	1.87e+06	6.41e+04	6.42e+04	5.43e+03
gzip	-	-	*212.4*	1.79e+07	1.61e+07	1.61e+07	1.61e+07
gunzip	-	-	-	*1.91e+04*	1.92e+06	1.92e+06	1.87e+06
md5sum	-	-	-	-	*3.03e+04*	3.38e+04	6.56e+04
sha1sum	-	-	-	-	-	*1.67e+04*	6.57e+04
sort	-	-	-	-	-	-	*4.24e+03*

traces of another program

3. we can make the structure of the return value frequency models apparent using k-means clustering

For the first hypothesis, we use Manhattan distance as a similarity metric in order to compare each trace of the same process with the return value model of that process. In effect, we compare each return value frequency to the corresponding average frequency among all traces of that program. To evaluate the second hypothesis, we use the Manhattan distance between each process model. The base set for the return values consists of all return values exhibited by all processes that are analyzed over all their runs. Table 7.4.1 shows how each model for a variety of program types (we include a variety of programs, like sorting, hashing, simple I/O, and compression) stays consistent with itself under the same input conditions (smaller Manhattan distance) and different from models for each other program (larger Manhattan distance).

Each process has a particular variance with each trace quantified in the similarity value

between its model and the trace itself, but when compared against the rest of the processes it can be easily distinguished. We ran each program ten times under the same input profile to collect the traces and generate the model for each program. We used as input profiles generic files/strings that can be easily replicated (in some cases no input was needed): date - N/A, echo - "Hello World!", gzip - *httpd-2.2.8.tar*, gunzip - *httpd-2.2.8.tar.gz*, md5sum *httpd-2.2.8.tar*, sha1sum - *httpd-2.2.8.tar* and sort - *httpd.conf* (the unmodified configuration file for httpd-2.2.8).

7.4.2 Clustering with k-means

Section 7.3 shows how to build profiles that are useful in predicting return values. The analysis here aims to achieve a better idea of what those actual return values are and how frequently real applications use them. We cluster the return values into frequency classes, and we chose the k-means method to accomplish the clustering. The large disparity in the magnitude of return value frequencies can reduce our ability to convey information about the overall structure of the model if displayed in a simple histogram. Accordingly, we found a new way to simultaneously display both the internal structure of each cluster as well as the external relationships between the clusters. Our clustering method captures the localized view of return value frequency per RV region and our visualization method provides insight into the relative coverage of a particular RV region. Figures 7.7, 7.8(a), 7.8(b), 7.9(a), 7.9(b), 7.10(a), and 7.10(b) present the clusters obtained for each of the analyzed processes. Each model has a predominant cluster (*e.g.,* cluster2 for data, cluster2 for gzip, *etc.*) which contains the discriminative return value frequencies and has coverage. The remaining clusters contain the lower frequency return values. We conjecture that by increasing the number of cluster we can achieve better granularity that can distinguish between different types of return values and classify them accordingly.

7.5 Limitations

STEM relies on Pin to reliably detect returns from a function. Detecting function exit is difficult in the presence of optimizations like tail recursion. Also, since the generated

profile is highly binary-dependent, STEM should recognize when an older profile is no longer applicable (and a new one needs to be built), *e.g.,* as a result of a new version of the application being rolled out, or due to the application of a patch. Finally, we plan to use this profiling capability to observe how well subsequent configurations of the execution profile match the expected configuration. Keeping track of the behavior of the application *after* error virtualization and repair is the essence of repair validation, and it also helps detect abnormal behavior due to the continuing existence of a compromise or fault despite the self-healing action.

7.6 Summary

Our modeling techniques can help select appropriate error virtualization values, inform the choice of rescue points, or drive the creation of repair policy templates. In addition, we provide a survey of return values used in real software. Finally, we propose *relative scaled k-means clusters*, a new way to simultaneously visualize both the micro and macro structure of feature-frequency behavior models.

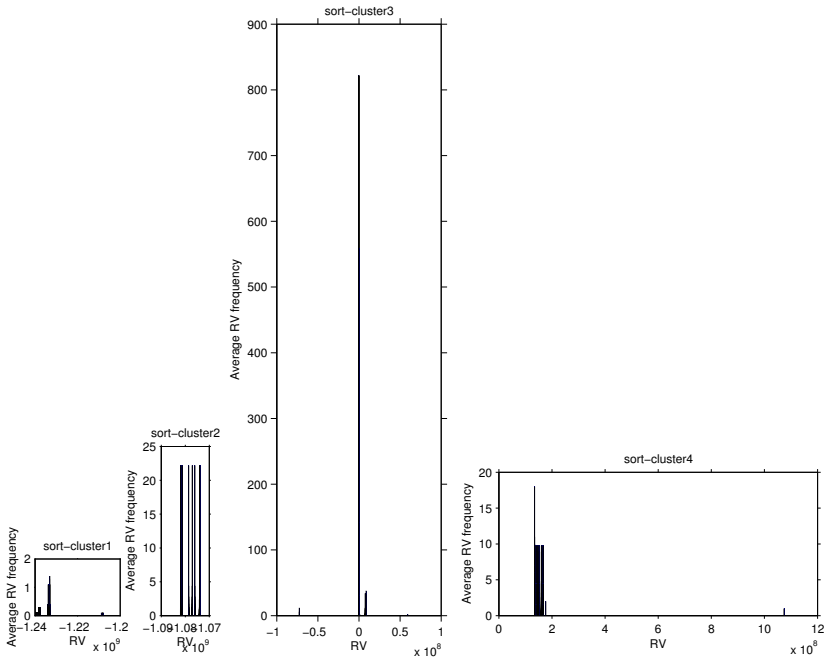
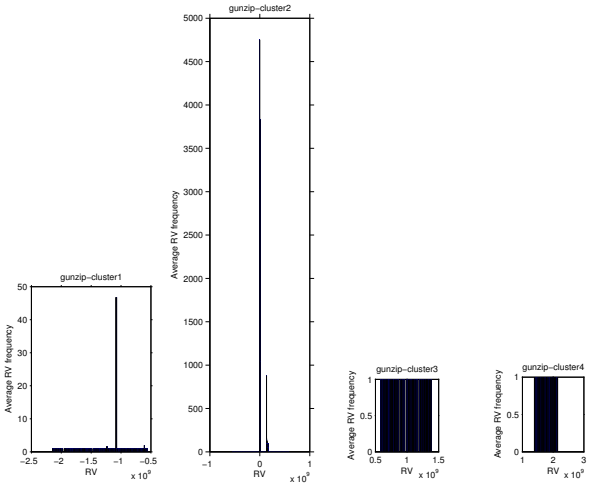
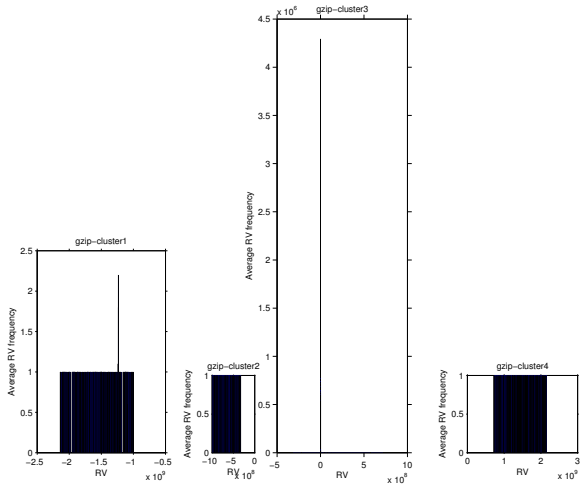


Figure 7.7: *Relatively Scaled k-means Clusters for* `sort`. Note how each component of the model is scaled relative to the others while displaying the distribution of similar-frequency values internally; this technique clearly displays the differences between the high frequency return values (cluster 3) and the frequent but more widespread parts of the model (cluster 4) as well as the behavior of values within each component. Both the vertical and horizontal dimensions of each cluster are scaled. We display the clusters in increasing order from left to right determined by the lower end of the horizontal axis range.

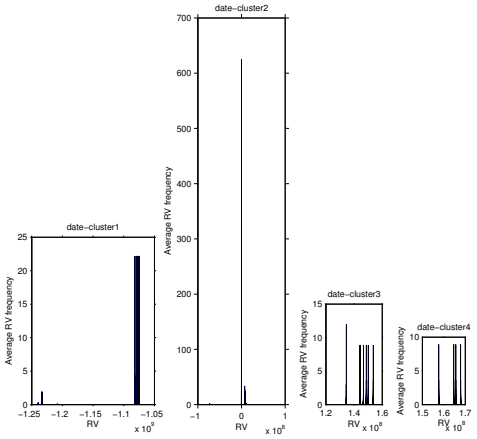


(a) *k-means cluster for* `gunzip` *return values.*

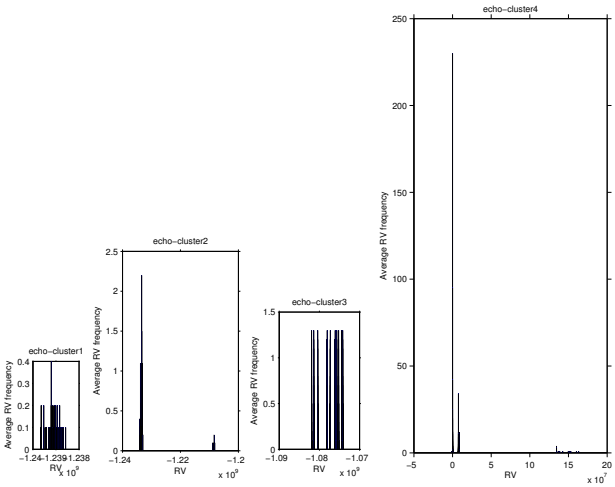


(b) *k-means cluster for* `gzip` *return values.*

Figure 7.8: Return Value Frequency Distributions for a Compression Program

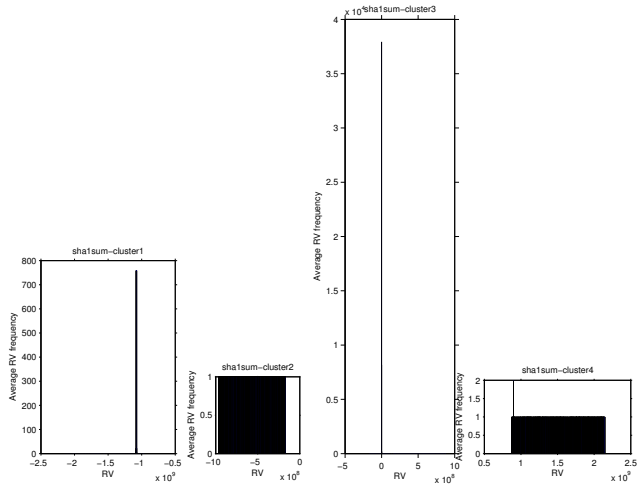


(a) *k-means cluster for* `date` *return values.*

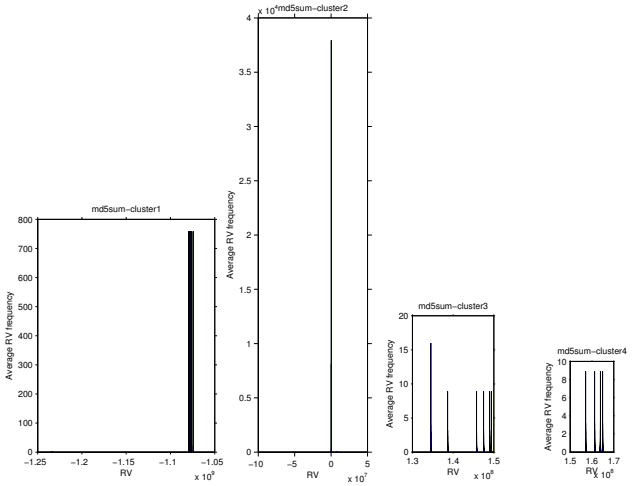


(b) *k-means cluster for* `echo` *return values.*

Figure 7.9: Return Value Frequency Distributions for Output Programs



(a) *k-means cluster for* `sha1sum` *return values.*



(b) *k-means cluster for* `md5sum` *return values.*

Figure 7.10: Return Value Frequency Distributions for Hash Programs

Chapter 8

Evaluation

The purpose of this evaluation section is twofold: determine the performance cost of our runtime supervision environments and characterize the benefit of our repair policy capabilities. As evidence of utility, we demonstrate the system's operation on real software and for real memory corruption vulnerabilities, in addition to a set of example programs and synthetic vulnerabilities. In this chapter, we demonstrate how to construct repair policy for real vulnerabilities and examine the performance of STEMv1 and STEMv2 with both informal microbenchmarks as well as the standard SPEC CINT2000 benchmark suite.

8.1 Microspeculation

The purpose of this section is twofold. We first examine with STEMv2 the tradeoff of performance vs. coverage: that is, for a series of real applications, how much of the execution we can cover at a given performance budget. We then examine the performance impact of STEMv2 and STEMv1. The underlying idea here is to provide some notion of the amount of work and resources required to microspeculate a particular slice of a program.

8.1.1 Coverage vs. Performance

We implemented a profiling mode for STEMv2. The profiling mode collects data on each invoked function in a dynamic trace of an application. This data is meant to measure the amount of "work" represented by a function or routine. In turn, this work and how it is

distributed among a software application's overall execution can help drive coverage policies and predict the potential performance slowdown due to supervision. This "work" quantity can be derived from a few different runtime measures:

- the number of machine instructions executed by the function
- the number of machine instructions that perform a memory write
- the cumulative size of all memory writes during the routine
- the amount of real time that a function takes to execute, as measured by STEM's timing machinery (itself based on the `x86 rdtsc` instruction)

We collect data for each of these measures using STEM's profiling mode on a number of applications. The timing information is cumulative; that is, it includes the time for children functions to execute. The timing information is collected with the precise `rdtsc` mechanism. The cumulative size of memory writes can help determine the memory log resources required for rollback and replay.

Each measure has advantages and drawbacks. For example, while instruction count is an intuitive unit and is straightforward to measure, there is a clear difference in computation between 100 logical "AND" operations and 100 floating point "DIV" operations (everything else, like data dependencies and structural hazards, being equal). In addition, the dynamic number of instructions is a parameter that is affected by the input (*i.e.*, network, file, time, signals, events) to the system and may fluctuate over time. A routine that looks "small" (in that it contains only a few tens of assembly instructions) may actually represent a great deal of work because those instructions implement a commonly used loop that is frequently exercised by the "normal" use of the system. On the other hand, using only timing information can obscure the effects of nondeterminism or interaction with other systems even though it may provide a more realistic sense of system response or throughput. Timing, however, is another dynamic measurement of the amount of work in a system; it is something affected by the input to the system. Finally, the cumulative size of all memory writes within a routine or slice can be a good indication of the amount of space resources

Table 8.1: *List of Profiling Microbenchmarks.* Command line utilities we extract workload distribution profiles from.

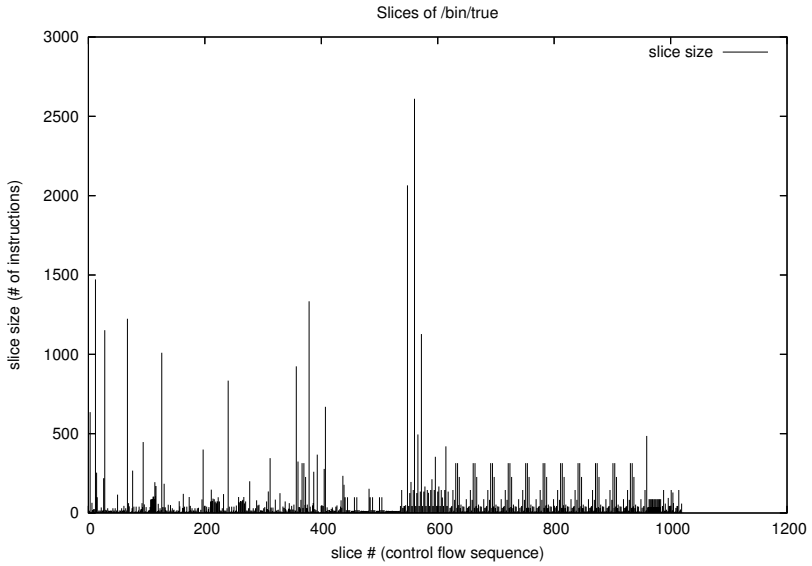
arch	basename /boot/vmlinuz-‘uname -r‘	cat /etc/passwd
cp /etc/passwd /tmp/passwd	date	df -lh
dmesg	dumpkeys	echo “hello, world.”
env	false	grep root /etc/passwd
uname -a	uname -r	hostname
ls /bin	mkdir /tmp/stem-tmp	netstat
sort /etc/passwd	ps	ps aux
pwd	rm /tmp/passwd	rmdir /tmp/stem-tmp
sleep 1s	sleep 10s	sleep 1m
rpm -qa	sync	true

STEM needs to supervise the slice. Again, confidence in the characteristic value range for this parameter depends on the quality of the training or normal profile.

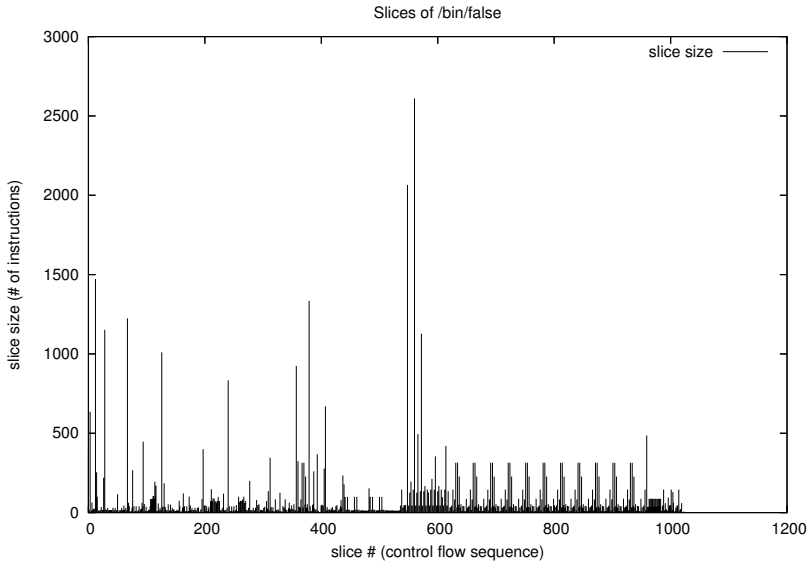
8.1.1.1 Slice “Work” Measures

We begin with a gentle introduction to the slice measures using some sample programs that are simple and deterministic in nature. We then examine some other programs under reproducible behavior profiles. We ran a series of simple benchmarks using a selection of the command line utilities included in the `/bin` directory of a Fedora Core 3 install. A list of the benchmarks and their invocations are presented in Table 8.1. The benefit of this list is that the experiments are easily reproducible, even though the specific work distribution (in terms of slice size) may not represent the work distribution in server or client applications that are the frequent target of attacks. The following figures (Figures 8.1, 8.2, 8.4(a), and 8.4(b)) show some of the interesting measures from the simpler benchmarks in this list.

If we examine the execution of `/bin/true` and `/bin/false` in terms of the number of instructions executed per slice, we can see that the work profile is very nearly the same, as



(a) Work Characterization for `/bin/true`



(b) Work Characterization for `/bin/false`

Figure 8.1: *Work Measurement of Slices Using Number of Executed Instructions.*

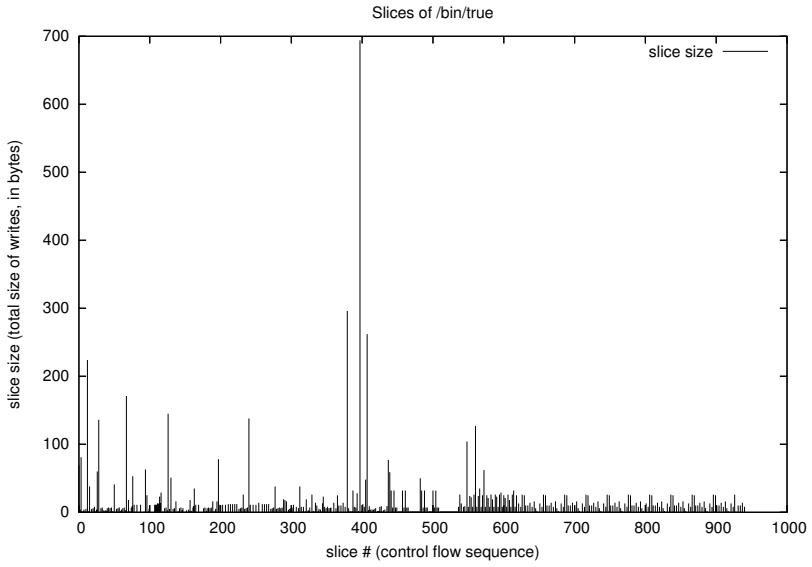


Figure 8.2: *Work Measurement of Slices Using Total Write Size.* This graph shows the total size, in bytes, of all writes to memory during each slice of `/bin/true`.

Figure 8.1 shows. These two programs are almost identical; they simply return zero and one, respectively. Each slice in the figure represents the dynamic occurrence of a function. These programs have about 1000 slices because of tracing the C library, not just their core functions.

As mentioned previously, the total number of executed instructions may not provide a good estimate of "work" in some cases, so we can also collect the total size of all memory writes made in each slice (*i.e.*, function) for each slice, as shown for `/bin/true` in Figure 8.2.

We can see, however, that there is a rough correspondence between the sections of this profile. In particular, there seems to be a startup preamble followed by a spike of work and finally a plateau as the program finishes and exits. When we plot a regression as shown in Figure 8.3, we see a fairly strong correlation between the number of instructions in a slice and the total size of memory writes made during that slice. Disregarding outliers, the regression coefficient is 0.79; with outliers included, the coefficient is a scant 0.25.

As expected, workload distributions differ among the microbenchmark programs from Table 8.1. We examine the workload characterization for `/bin/arch` (a utility that displays

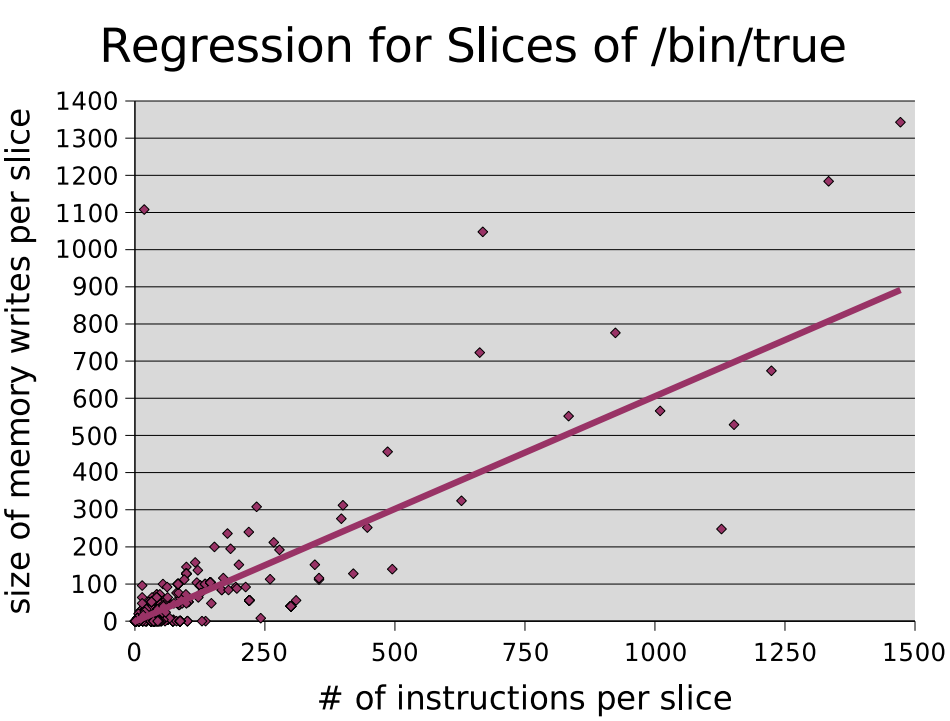
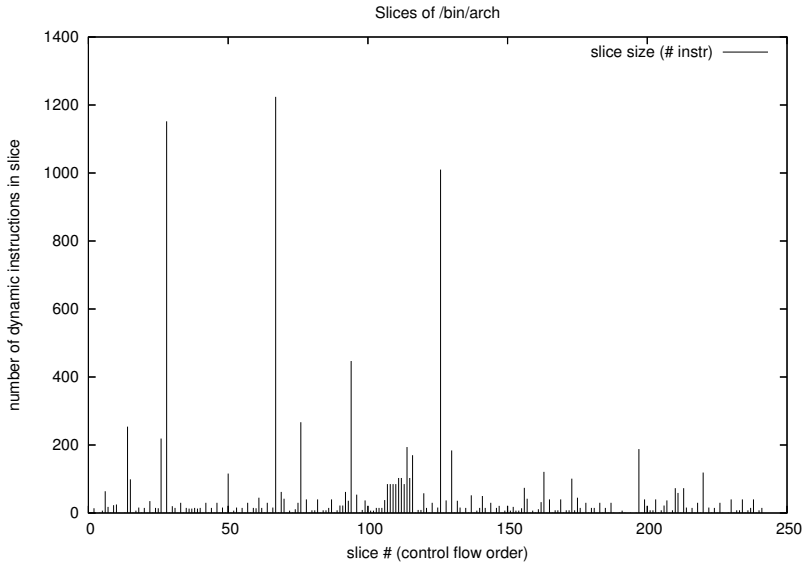
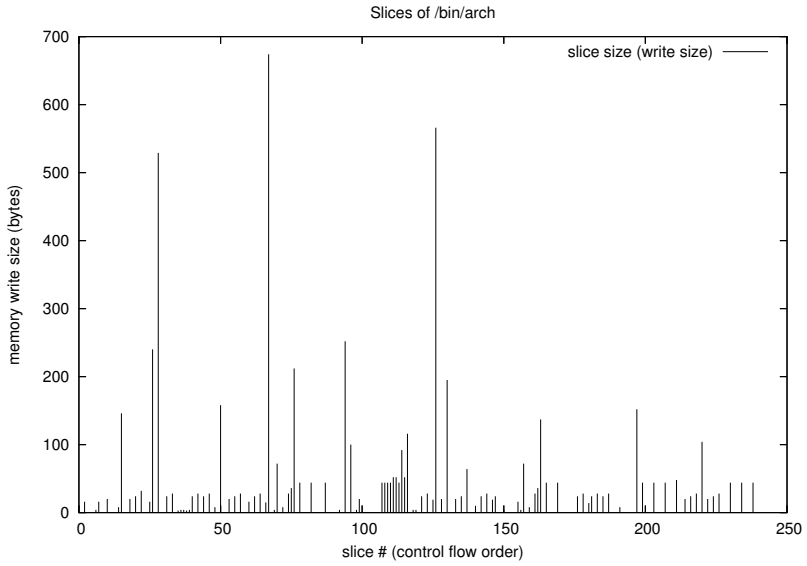


Figure 8.3: *Relationship Between Instructions per Slice and Total Memory Write Size per Slice for* `/bin/true`. The linear regression coefficient is 0.79, which indicates a strong relationship between the number of instructions executed in each slice and the total size of memory written in that slice. While we expect this to be true for most programs, more compute–intensive programs that do not interact with memory much may have a reduced coefficient.

the machine architecture, *e.g.*, `x86`) displayed in Figure 8.4. The figure shows both the total number of instructions executed during each slice of `/bin/arch` and the total size, in bytes, of all writes to memory during each slice. We observe that these distributions differ from those of `/bin/true` and `/bin/false`. The different distributions for each microbenchmark provide two things: a sanity check on our profiling instrumentation, and a general sense that most slices of any given application contain a relatively low level of "work": only a few slices or routines represent a substantial amount of work; the overhead of monitoring (in terms of added execution time and additional memory resources) will be felt most in those routines.



(a) Instructions per Slice



(b) Cumulative Memory Write Size per Slice

Figure 8.4: *Work Characterization for* `/bin/arch`.

Figure 8.5 shows memory write size graphs for a selection of the other microbenchmarks from Table 8.1. The main point here is that we provide a user of STEM with the capability to examine their application under some use or input profile to discover where the most work is done in a "normal" run of the application; such information can help them plan and provision the resources they need to run the application under STEM with a particular coverage policy. For example, if STEM users have confidence that a particular costly and regularly invoked routine does not include a vulnerability, then they can avoid instrumenting that function.

8.1.1.2 Macrobenchmarks

While the microbenchmarks provide both a sanity check on STEM's profiling machinery and a glimpse into the workload distributions for some commonly used command-line applications, we would like to see if we can discover the work distribution for mixed workloads and larger software systems. We begin by comparing two shell sessions for mixed workloads; each session issues a number of different commands. We then examine two compute-intensive applications, `md5sum` and `gzip`. Finally, we examine the behavior of Apache as it serves a recursive examination of the installation manual.

Using STEM to supervise a shell session reveals some interesting phenomena. After STEM starts up, there is a noticeable two or three second lag after the first command is issued as that new code is instrumented; subsequent invocations do not experience the lag. This result matches our expectations based on STEM's instrumentation method (dynamic binary rewriting), in which code is re-written when it is first encountered, but then executes at machine speed, albeit with the overhead introduced by the inserted instrumentation. Both Figure 8.6(a) and Figure 8.6(b) distinctly show very similar startup phases followed by a quiescent period and then the initial instrumentation of the command forking code, followed by the actual session execution: the invocation of shell commands; the discontinuities correspond to different commands and changes in the profile.

Our second set of macrobenchmarks examines the behavior of two compute-intensive applications (`md5sum` and `gzip`). For the sake of reproducibility, we operate these utilities on a tarball of a recent version of Apache (httpd-2.0.54), a Fedora Linux kernel image, and

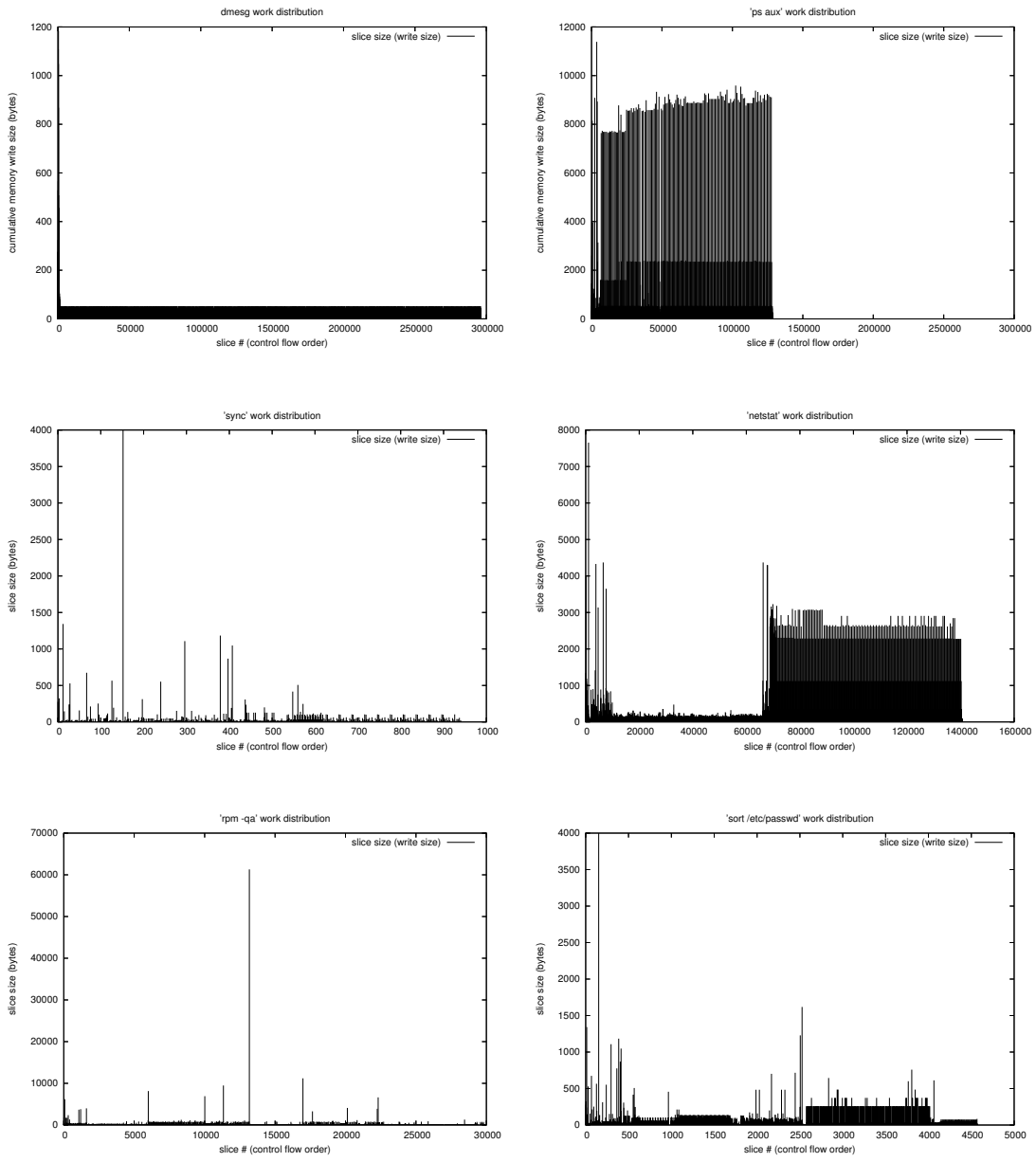
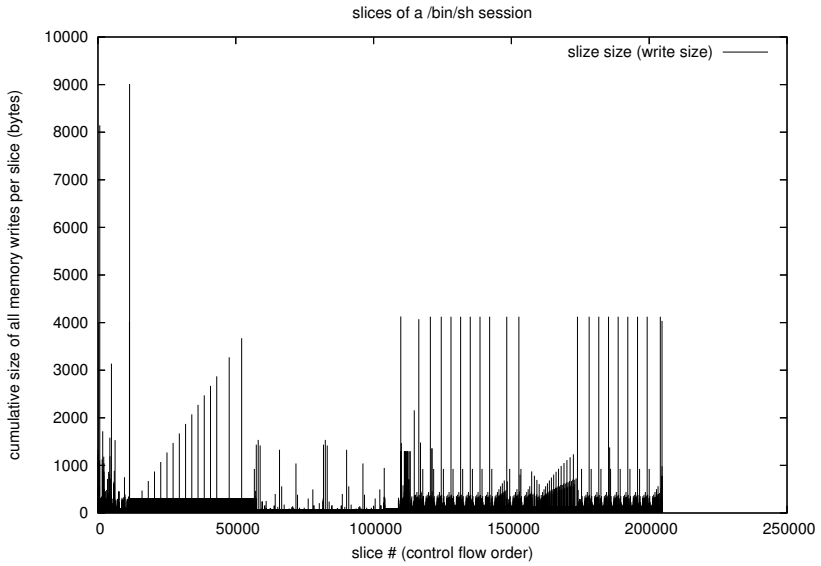
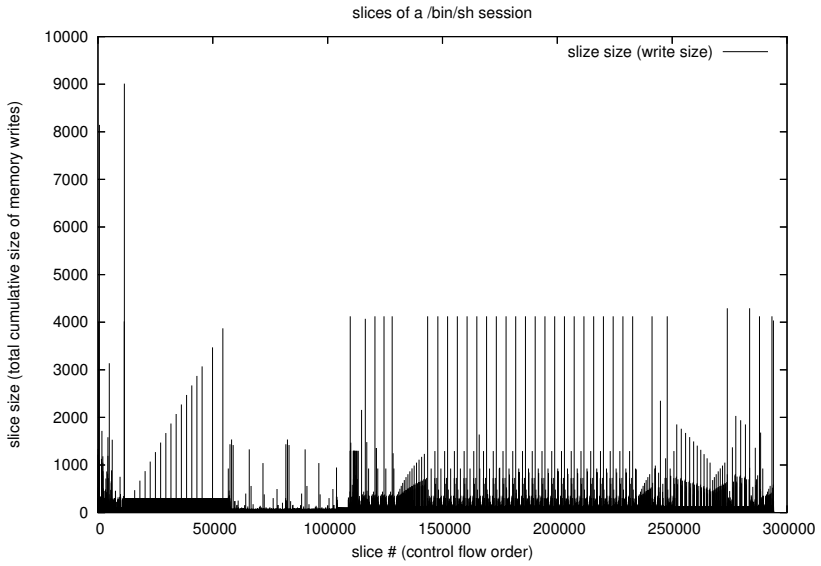


Figure 8.5: *Work Distribution for Various Utilities.* We measure `dmesg`, `rpm`, `netstat`, `sync`, `sort`, and `ps`. Each has a distinct profile (in these graphs, the cumulative size of memory writes during each slice).



(a) A `/bin/sh` Session with Repeated Invocations of `ls`



(b) A `/bin/sh` Session with a Different, Longer Set of Commands

Figure 8.6: *Work Distribution for Two shell sessions*

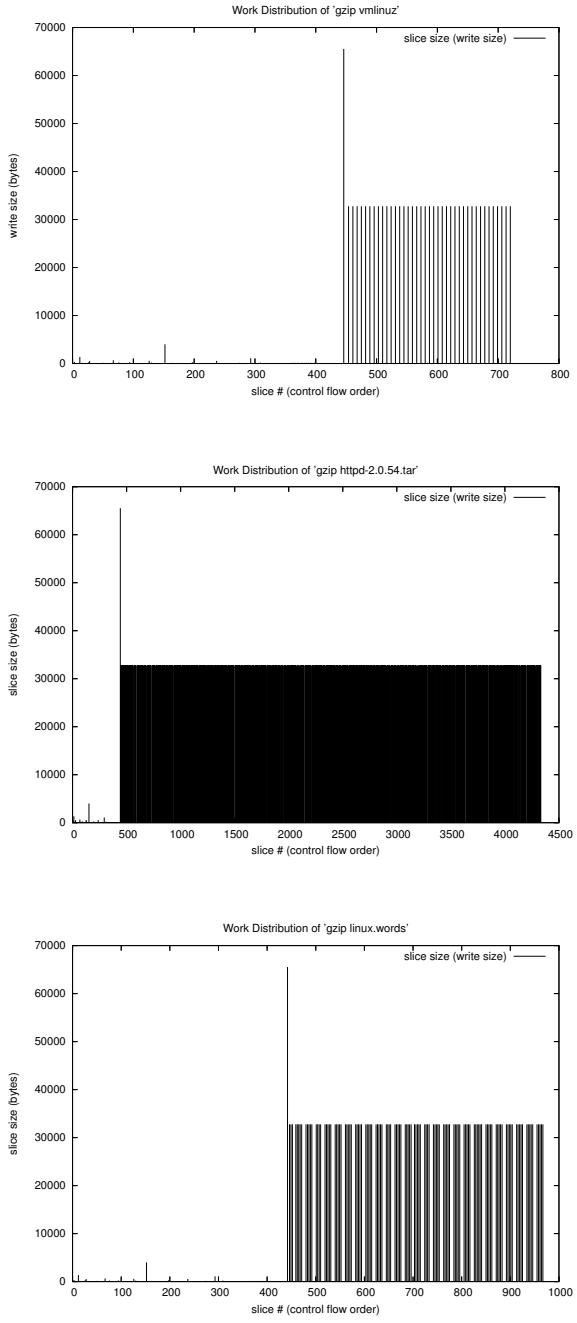


Figure 8.7: *Work Distribution for* `gzip`. The gzip utility's operation on a kernel image, an Apache tarball, and a dictionary.

the Linux dictionary provided with Fedora Core 3. These are the same files used in the experiments detailed in Section 8.1.3. The results for `gzip` are plotted in Figure 8.7, and the results for `md5sum` are plotted in Figure 8.8.

We first note that the behavior profiles, although they display a marked contrast between the behavior of `gzip` and the behavior of `md5sum`, are internally consistent: `gzip` shows a low level of startup activity (in terms of memory writes) followed by a spike and then a plateau of memory write size (at about 32KB, which may make sense as the default size of some internal `gzip` buffer). Note that the horizontal axis of Figure 8.7(b) is much longer than either Figures 8.7(a) or Figure 8.7(c). This phenomena makes sense since the tar archive of Apache is much larger than either the kernel image or the dictionary. Disregarding this difference, all three invocations show the same type of behavior, and the smaller resolution of the latter two graphs shows periodic drop-offs during the plateau.

In turn, `md5sum` displays a dominant plateau (at about 11KB) lasting most of the execution, with a small startup spike and a much smaller plateau of cleanup activity. Even though we may think of `md5sum` as being a utility that only reads information, this behavior does make sense: the memory writes that STEM tracks are at the machine instruction level, and the micro-operations of the MD5 algorithm need to (at the very least) store the hashing data for further calculation within a round. Equally telling is that the work distribution graphs (not displayed) for `md5sum` using number of executed instructions as the measure of work show the same block behavior, as we might expect from a compute-intensive workload executing the same rounds of instructions.

Finally, we examine Apache's workload distribution given a client that performs a recursive descent of the Apache manual pages. We run Apache in single-threaded mode; over a period of 5 minutes, it serves roughly 13MB of data at a rough rate of 5 requests per second. Figure 8.9 displays the results.

8.1.1.3 Coverage Summary

The lessons we have learned from this examination of workload distribution is that workloads vary widely with individual applications. Furthermore, for different uses of the same application (*e.g.*, the shell experiments) the workload can vary depending on the input pro-

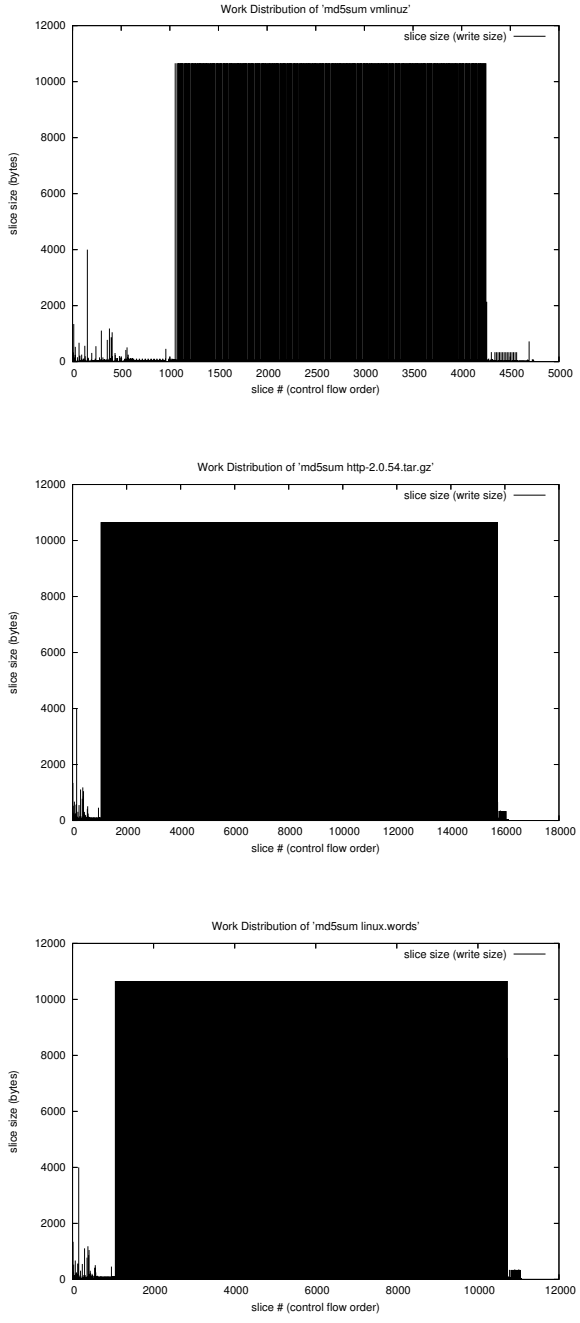
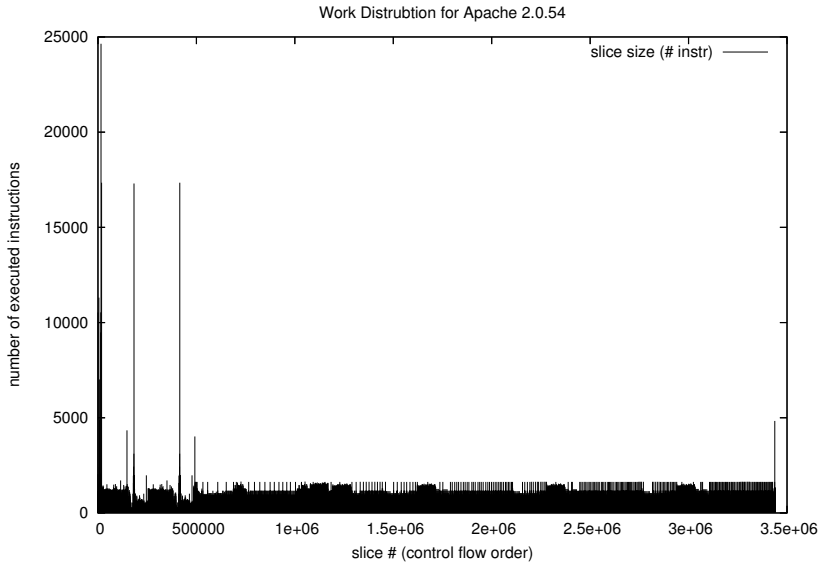
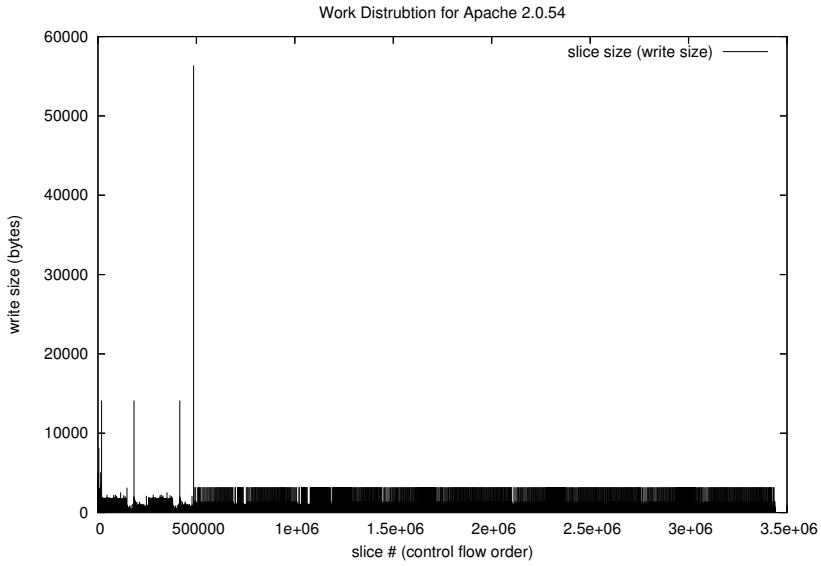


Figure 8.8: *Work Distribution for* `md5sum`. The md5sum utility's operation on a kernel image, an Apache tarball, and a dictionary. Note that this profile is good news from a cryptanalysis perspective.



(a) Apache's Instructions Executed



(b) Apache's Memory Writes

Figure 8.9: *Work Distribution for Apache.* A client (*i.e.*, `wget`) performs a recursive descent of the manual pages that come with the default Apache install. Here, we show both the "instructions executed" and "size of memory writes" profiles for Apache.

file. These lessons should be no surprise; they are intuitive. We provide concrete evidence of this knowledge as a way to demonstrate that, despite the performance impact of STEM (illustrated below), STEM also provides a way for users to see what parts of their application will incur a relatively hefty performance penalty because of its supervision. In essence, STEM magnifies the performance overhead of existing code. STEM, however, provides a way for users to discover these workload distributions and adjust their coverage policy or resource allocation. While STEMv2, at least, significantly reduces the performance penalty for using microspeculation (as we will see in the following sections), we cannot entirely eliminate it; we do not, however, leave the user in the dark: we provide a seamless mode of operation for discovering these obstacles.

The microbenchmarks and macrobenchmarks provide some insight into the overhead of STEM. Because STEM is a compute-intensive workload (it inserts additional instructions for each machine instruction), future efforts can see what kind of impact STEM has on the mixed workloads represented by the parts of the SPEC CINT2000 benchmark suite.

8.1.2 Performance of STEMv1

As we explain in Chapter 4, STEMv1 is an emulation–based approach derived from the Bochs `x86` full-system emulator. As such, it has a considerable performance impact on the applications it supervises. This performance impact is somewhat offset by the efficacy of its primary detection technique (ISR) and the ability to cover "weak" portions of the execution stream, as suggested by Section 4.1.

We evaluated the performance impact of STEMv1 by instrumenting the Apache Web server and performing microbenchmarks on some shell utilities. We chose the Apache *flood httpd* testing tool to evaluate how quickly both the non-emulated and emulated versions of Apache would respond and process requests. In our experiments, we chose to measure performance by the total number of requests processed, as reflected in Figure 8.10. The value for total number of requests per second is extrapolated (by *flood*'s reporting tool) from a smaller number of requests sent and processed within a smaller time slice; the value should not be interpreted to mean that our Apache instances actually served some 6000 requests per second.

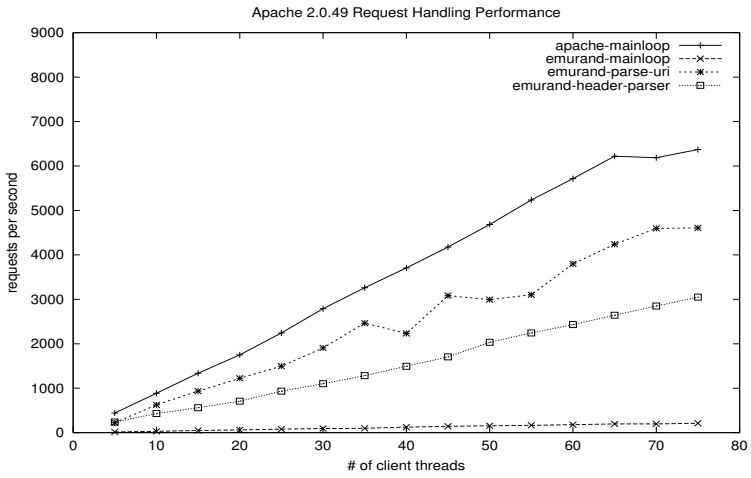


Figure 8.10: *Performance of STEM under various levels of emulation.* While full emulation is fairly expensive, selective emulation of input handlers appears quite sustainable. The "emurand" designation indicates the use of STEM (emulated randomization).

We selected some common shell utilities and measured their performance for large workloads running both with and without *STEM*. For example, we issued an '*ls -R*' command on the root of the Apache source code with both *stderr* and *stdout* redirected to */dev/null* in order to reduce the effects of screen I/O. We then used *cat* and *cp* on a large file (also with any screen output redirected to */dev/null*). Table 8.2 shows the result of these measurements. As expected, there is a large impact on performance when emulating the majority of an application. Our experiments demonstrate that only emulating potentially vulnerable sections of code offers a significant advantage over emulation of the entire system.

8.1.3 Performance of STEMv2

The goal of our performance evaluation is to characterize STEM's impact on the normal performance of an application. We do so by performing a series of microbenchmarks. These experiments help characterize STEM's impact on both startup time and normal operation. STEM incurs a relatively low performance impact for mainline execution of real-world software applications, including both interactive desktop software as well as server programs. The overhead includes both STEM's core operations (memory logging, repair policy inter-

Table 8.2: Microbenchmark performance times for various command line utilities.

Test Type	trials	mean (s)	Std. Dev.	Min	Max	Instr. Emulated
ls (non-emu)	25	0.12	0.009	0.121	0.167	0
ls (emu)	25	42.32	0.182	42.19	43.012	18,000,000
cp (non-emu)	25	16.63	0.707	15.80	17.61	0
cp (emu)	25	21.45	0.871	20.31	23.42	2,100,000
cat (non-emu)	25	7.56	0.05	7.48	7.65	0
cat (emu)	25	8.75	0.08	8.64	8.99	947,892

pretation, *etc.*) and Pin's instrumentation and binary rewriting machinery.

Although the time it takes to self-heal is also of interest, our experiments on synthetic and real vulnerabilities show that this amount of time depends on the complexity of the repair policy (*i.e.*, how many memory locations need to be adjusted) and the memory log rollback. Even though rollback is an $O(n)$ operation, STEM's self–healing and repair procedure usually takes well under a second (using the `x86 rdtsc` instruction we observe an average of tens of milliseconds) to interpret repair policy for these vulnerabilities.

8.1.3.1 Experimental Setup

In the microbenchmarks that we discuss next, we used multiple runs of applications that are representative of the software that exists on current Unix desktop environments. We tested `aplay`, Firefox, `gzip`, `md5sum`, and `xterm`, along with a number of smaller utilities: `arch`, `date`, `echo`, `false`, `true`, `ps`, `uname`, `uptime`, and `id`. The applications were run on a Pentium M 1.7 GHz machine with 2 GB of memory running Fedora Core 3 Linux. We used a six minute and ten second WAV file to test `aplay`. To test both `md5sum` and `gzip`, we used three files: *httpd-2.0.53.tar.gz*, a Fedora Core kernel (*vmlinuz-2.6.10-1.770_FC3smp*), and the */usr/share/dict/linux.words* dictionary. Our Firefox instance simply opened a blank page. Our `xterm` test creates a terminal window and executes the `exit` command. We also tested two versions of Apache (2.0.53 and 2.2.4) by attaching STEM after Apache starts and using `wget` to download the Apache manual from another machine on the same

network switch. Doing so gives us a way to measure STEM's impact on normal performance excluding startup (shown in Table 8.4). In the tables, the suffixes for gzip and md5sum indicate the kernel image (k), the httpd tarball (h), and the dictionary (d). During these tests, no specific repair policy was employed (although policy evaluation hooks were in place).

8.1.3.2 Microbenchmarks

Of general concern is whether or not STEM slows an application down to the point where it becomes apparent to the end-user. We have not performed a user study, so the following observations are ours. Even though STEM has a rather significant impact on an application's startup time (as shown in Table 8.3), it appears to not have a human-discernible impact when applied to regular application operations. For example, Firefox remains usable for casual web surfing when operating with STEM. In addition, playing a music file with `aplay` also shows no sign of sound degradation – the only noticeable impact comes during startup. Disregarding this extra time, the difference between `aplay`'s native performance and its performance under STEM is about 2 seconds. If STEM is attached to `aplay` after the file starts playing, there is an eight second delay followed by playback that proceeds with only a 3.9% slowdown. Most of the performance penalty shown in Table 8.3 is exaggerated by the simple nature of the applications. Longer running applications experience a much smaller impact relative to total execution, as seen by the `gzip`, `md5sum`, and Firefox results.

Although most applications suffer a hefty performance hit, the majority of the penalty occurs during application startup and exit. In Table 8.3, we remove a well-defined portion of the application's initialization from the performance consideration. Removing supervision of this portion of the startup code improves performance over full supervision. The remaining time is due to a varying amount of startup code, the application itself, and exit code.

In order to measure STEM's impact on an application, and thereby eliminate application startup from consideration, we perform a series of other measurements, such as attaching to Apache after its initialization has completed (as shown in Table 8.4). The `aplay` performance results are also revealing, and we conduct measurements of `sshd`, `xboard`, and `scp` to remove the startup penalty from consideration.

Table 8.3: *Performance Impact Data*. Attaching STEM at startup to dynamically linked software incurs a significant performance penalty. In the rightmost two columns, we present a revised slowdown (rSlowdown) and execution times (rSTEM) after removing a well-defined portion of the startup.

Program	**Native (s)**	**STEM (s)**	**Slowdown**	**rSTEM (s)**	**rSlowdown**
`aplay`	371.0125	459.759	0.239	—	—
`arch`	0.001463	14.137	9662.021	3.137	2143.22
`xterm`	0.304	215.643	708.352	194.643	639.273
`echo`	0.002423	17.633	7276.342	5.633	2323.803
`false`	0.001563	16.371	10473.088	4.371	2795.545
`true`	0.001552	16.025	10324.387	5.025	3236.758
`Firefox`	2.53725	70.140	26.644	56.14	21.128
`gzip-h`	4.51	479.202	105.253	468.202	102.814
`gzip-k`	0.429	58.954	136.422	47.954	110.780
`gzip-d`	2.281	111.429	47.851	100.429	43.025
`md5-k`	0.0117	32.451	2772.589	20.451	1746.948
`md5-d`	0.0345	54.125	1567.841	42.125	1220.014
`md5-h`	0.0478	70.883	1481.908	58.883	1230.862
`ps`	0.0237	44.829	1890.519	31.829	1341.996
`uname`	0.001916	19.697	10279.271	8.697	4538.144
`uptime`	0.002830	27.262	9632.215	15.262	5391.932
`date`	0.001749	26.47	15133.362	14.47	8272.299
`id`	0.002313	24.008	10378.592	13.008	5622.865

8.1.3.3 Performance Without Startup Penalty

STEM is currently implemented using Pin's "Routine" RTN_* API instrumentation. We did so initially to ease the difficulty of implementation. It appears that this type of instrumentation statically instruments all functions in a binary image as that image is loaded, whether or not that function is ever actually encountered during execution. As a result, in applications that contain large libraries of code, such as applications linked against GTK or X libraries, those GUI libraries are also needlessly instrumented. For short running command-line applications like many of those in Table 8.3, these programs are dynamically linked against the GNU C library, resulting in needless instrumentation as dynamic libraries are loaded and linked. We intend to switch to Pin's per-instruction INS_* API instrumentation to remove this performance penalty. We have already done so for a similar tool that tracks tainted information flows with a significant performance increase. In the following discussion, we try to characterize the performance impact of STEM without the startup penalty. We expect that with the use of the INS_* API, we can improve performance even more, in both the startup phase and in normal operation.

Table 8.4: *Impact on Apache Excluding Startup.*

Apache	**Native (s)**	**STEM (s)**	**Impact %**
`v2.0.53`	3746	6550	74.85%
`v2.2.4`	16215	27978	72.54%

In Table 8.3, note that `aplay` shows fairly good performance; a roughly six-minute song plays in STEM for 88 seconds longer than it should — with 86 of those seconds coming during startup, when the file is not actually being played. In addition, we played a chess game with the `xboard` GUI to `gnuchess` under STEM. It took 533 (roughly 8 minutes) seconds to play the game[1], executing 14631030 instructions, 3248180 write operations, and 783969 functions. The GUI experienced a slow start, and first-time manipulation of board

[1]The actual time spent probably says more about the skill level of the human involved rather than STEM's impact. We provide the timing information as another example that STEM's impact is not pathologic: a short chess game does not take an exorbitant amount of time when played under STEM.

pieces and menu selections was slow. After this, however, play proceeded at a pace that could not be distinguished from normal operation.

The difference between the startup penalty and the impact on post-startup operation is also evident when we test the OpenSSH `sshd` in debug mode (so that it remains single-threaded and does not run as a daemon). To remove human reaction time from the measurements, we set up authentication using SSH keys and the `ssh-agent`. We invoke the client with the -v option and execute the `exit` command. Although `sshd` startup (47046 functions and about 4.5 million instructions) takes longer under STEM (58 seconds vs. 0.85 seconds — a slowdown of 68X), the execution of the client interaction with the server takes 1.3 seconds vs. 0.229 seconds. In addition, when we use the shell in an interactive manner, there is no perceptible difference in execution performance for a variety of command-line applications. Finally, we use STEM to supervise the execution of `scp` performing a transfer of a 7.4 MB file between a Canadian cable ISP and the Columbia CS Department. This test allows us to reduce the relative importance of the startup penalty. Native execution of `scp` takes 130 seconds to transfer the file. Execution under STEM accomplishes the transfer in 148 seconds (166 seconds with startup) using 2513998 instructions in 50726 routines. This difference represents a 1.13X overhead.

We tested STEM's impact on two versions of Apache by starting Apache in single-threaded mode (to force all requests to be serviced sequentially by the same thread). We then attach STEM after verifying that Apache has started by viewing the default homepage. We use `wget` to recursively retrieve the pages of the online manual included with Apache. The total downloaded material is roughly 72 MB in about 4100 files. STEM causes a 74.85% slowdown, far less than the large factor when including startup. Native execution of Apache 2.0.53 takes 0.0626 seconds per request; execution of the same under STEM takes 0.1095 seconds per request. For a newer version of Apache, we observe an improvement to 72.54%.

In addition to the variety of microbenchmarks we have used to evaluate STEMv2's performance impact, we ran the SPEC CINT2000 benchmark with STEM on two machines: a dual CPU Xeon with 1GB of primary memory, and a single CPU Pentium M with 2GB of primary memory. The SPEC benchmark consists of a number of applications, including `gzip`, `gcc`, `bzip2`, `vpr` (a FPGA circuit placement and routing tool), `mcf` (a combinatorial

optimization program), `crafty` (a chess engine), `parser` (a word processor), a Perl benchmark that manipulates email archives, `vortex` (an object-oriented database), and `twolf` (a route simulator).

We had two controls for this experiment, one on each hardware platform. The experiments were performed using STEM to execute each run of each benchmark in the suite. The applications were not recompiled or altered in any way. In the Xeon control and test case, only a single CPU was used in order to refrain from biasing the results.

Our purpose in conducting these tests was not to show an artificially positive view of STEM's performance given what could be viewed as two over-provisioned test platforms (a hyperthreaded dual CPU server platform with a generous amount of primary memory and an efficient CPU platform with an even more generous amount of primary memory). Instead, the comparison is between the normal performance of the machines and STEM's performance running the same tests on those same machines. Thus, the relative performance impact is of interest rather than the absolute performance offered by STEM. In addition, we note that the entire SPEC benchmark runs in STEM, so startup costs (such as those elucidated in Table 8.3) are included, heavily degrading STEM's raw performance.

8.1.3.4 Summary

Most of the work done during startup loads and resolves libraries for dynamically linked applications. STEM can avoid instrumenting this work (and thus noticeably reduce startup time) in a few ways. The first is to simply not make the application dynamically linked. We observed for some small test applications (including a program that incorporates the example shown in Figure 5.2 from Section 5.3) that compiling them as static binaries reduces execution time from fifteen seconds to about five seconds. Second, since Pin can attach to applications after they have started (in much the same way that a debugger does), we can wait until this work completes and then attach STEM to protect the mainline execution paths. We used this capability to attach STEM to Firefox and Apache after they finish loading (we measured the performance impact on Apache using this method; see Table 8.4). Also, we can allow the application to begin executing normally and only attach STEM when a network anomaly detector (*e.g.*, Anagram [WPS06]) or some other IDS mechanism (such

as Snort) issues an alert based on the incoming traffic. Furthermore, as we have discussed in Section 8.1.3.3, we can switch our implementation from using Pin's RTN_* API to the INS_* API. Finally, it may be acceptable for certain long-running applications (*e.g.*, web, mail, database, and DNS servers) to amortize a lengthy startup time (on the order of minutes) over the total execution time (on the order of weeks or months).

8.2 Repair Policy

In this section, we show how to generate repair policy for a series of different types of vulnerabilities, ranging from the synthetic vulnerabilities we have used as running examples to real vulnerabilities in real software. We also evaluate the performance of the repair process itself by using the `rdtsc` mechanism to time STEM's operation of the repair.

8.2.1 Synthetic Vulnerabilities

As a gentle introduction to our testing of repair policy performance, we begin with some short example programs containing synthetic faults or vulnerabilities. Throughout the development of STEM, we used a few small example programs to test the functionality of STEM and repair policy. The idea with each is to illustrate an aspect of detection, repair, or survivability on a small scale. Such a low-level of complexity enables us to more easily observe, diagnose, and debug the interaction between STEM, Pin, the platform, the exploit input or conditions, and the application. In most of the following examples, some knowledge of the source code aids the construction of more appropriate policies.

8.2.1.1 Stack Overwrite Example

The first example program is fairly simple. It contains a function named `exploitme` with a loop that progressively writes the value `0xDEADBEEF` over the function's stack frame. Without STEM, the application terminates due to a segmentation violation. This violation occurs because the saved base pointer and the return address are overwritten. As the function returns, these locations are used to attempt restoration of the parent stack frame.

We wrote a small repair policy, shown in Figure 8.11, to measure the integrity of these

```
ivp MeasureStack :=:
   ('ebp      == 'shadowstack[1]),
   ('raddress == 'shadowstack[0]);

rp RepairStack :=:
   ('ebp      == 'shadowstack[1]),
   ('raddress == 'shadowstack[0]);

tp exploitme
 &MeasureStack
 :=:
 &RepairStack,
 (unroll);
```

Figure 8.11: *Example Repair Policy*

two items. If they are violated, we instruct STEM to undo all the memory changes made during that routine and restore the integrity of the stack frame. As the return value is not checked, we do not adjust it. The repair allows the application to avoid crashing and finish execution.

8.2.1.2 Apache Stack Overwrite Example

We also create a repair policy for the vulnerability we inserted into Apache for our FLIPS experiments. The vulnerability we injected is similar to the stack overwrite program above and is shown in Figure 8.12.

8.2.1.3 Authentication Skip Example

We constructed a simple command line utility to accept login credentials, vet them, and start a shell session. This utility contains code similar to that shown in Figure 5.2. The code contains an overflow of a fixed-sized local buffer in the `checkpassword` function. The semantics of the code require that a healed exploit of this vulnerability returns a value of 200. We constructed a repair policy similar to that shown in Figure 5.4, but with the value of the authentication failure code equal to 200.

A sample successful run of the application (with appropriate authentication credentials)

```
ivp MeasureStack :=:
   ('ebp      == 'shadowstack[1]),
   ('raddress == 'shadowstack[0]);

rp RepairStack :=:
   ('ebp      == 'shadowstack[1]),
   ('raddress == 'shadowstack[0]);

tp exploitMe &MeasureStack
 :=:
 ('rvalue == -1), &RepairStack, (unroll);

tp exploitMe_embedded &MeasureStack
 :=:
 ('rvalue == -1), &RepairStack, (unroll);

tp exploitMe_stackAssist &MeasureStack
 :=:
 ('rvalue == -1), &RepairStack, (unroll);

tp exploitMe_stackCorrupt &MeasureStack
 :=:
 ('rvalue == -1), &RepairStack, (unroll);
```

Figure 8.12: *Repair Policy for Apache Sample Vulnerability*

```
[michael@xoren testapps]$ ./authskip hello
login failed.
[michael@xoren testapps]$
```

Figure 8.13: *Invoking* `authskip` *with Incorrect Credentials*

is shown below. The program drops a shell (as it should). We simply exit the shell, and the program ends. Running the program with the proper credentials in STEM produces the same behavior, as shown in Figure 8.15.

```
[michael@xoren testapps]$ ./authskip michael
login OK.
sh-3.00> exit
exit
[michael@xoren testapps]$
```

Figure 8.14: *Invoking* `authskip` *with Correct Credentials*

On the other hand, if an attacker supplies data that would overflow the internal fixed-size buffer in `checkpassword`, such as the long sequence of "A" characters shown in Figure 8.16, then STEM will interpret the repair policy and prevent the program from creating the shell. Note that `c8` is the hexadecimal value of 200, indicating an unsuccessful login attempt. Repair takes approximately 22 milliseconds, and the login attempt fails. Running `authskip` alone with the same input causes a segmentation fault.

8.2.2 Wilander Testbed

The Wilander Testbed [WK03] consists of a variety of buffer overflows that attack the process memory space through manipulation of stack variables, function pointers, and other memory regions. Wilander and Kamkar [WK03] evaluate the efficacy of protection mechanisms like StackGuard and Propolice. Our purpose in using this testbed is to demonstrate the basics of constructing and testing simple repair policy for real vulnerability types. John Wilander provided access to the test suite, which contains 18 of the 20 attack cases examined in the paper.

The requirements for "healing" or "repairing" each test case are rather straightforward.

```
[michael@xoren testapps] stem ./authskip michael
STEM starting at 1193096821...
[RVM]: Ripple policy file is: /home/michael/.stem/conf/rpolicy.rpl
[RVM]: constraint solver located at: /home/michael/bin/realpaver
[RVM]: starting /home/michael/.stem/bin/wvm
parsing [/home/michael/.stem/conf/rpolicy.rpl]...
reached main() at 1193096824
reached init() at 1193096834
login OK.
sh-3.00> exit
exit
reached fini() at 1193096843

Done at:                  1193096845
Total Instructions:       99116
Total write operations:   13844
Speculation Depth   :     0
Total CoSAK routines:     997
Total number of routines: 1382
STEM finished with code 0
[michael@xoren testapps]
```

Figure 8.15: *Invoking* `authskip` *under STEM*

```
[michael@xoren testapps] stem ./authskip AAAAAAAAAAAAAAAAAAAAAAAA
STEM starting at 1193098119...
[RVM]: Ripple policy file is: /home/michael/.stem/conf/rpolicy.rpl
[RVM]: constraint solver located at: /home/michael/bin/realpaver
[RVM]: starting /home/michael/.stem/bin/wvm
parsing [/home/michael/.stem/conf/rpolicy.rpl]...
...
Return address or base pointer corrupted.
[STEM]: starting RP at 1193098134
...
***STEM: UNROLL MEMORY LOG BEGIN***
restored 311
***STEM: UNROLL MEMORY LOG END***
[STEM]: check_credentials() completing healing...
[STEM]: Overwrote EBP
check_credentials(): EAX at ADD,[0x8048563] = 0xc8
...
[STEM]: ending RP at 1193098134
[STEM]: healing takes 22855710 ns
login failed.
reached fini() at 1193098134

Done at:                  1193098135
Total Instructions:       101100
Total write operations:   13983
Speculation Depth   :     0
Total CoSAK routines:     1038
Total number of routines: 1398
STEM finished with code 0
```

Figure 8.16: *STEM Repairing* `authskip`

Table 8.5: *Wilander Testbed Index*. The first group contains buffer overflows on the stack all the way to the target. The second group contains heap/BSS overflows all the way to the target. The third group contains overflows of a pointer on the stack that then points to the target. The last group contains overwrites of a heap/BSS pointer that then points to the target.

Test Case Index	**Overwrite Target**
-4	Parameter function pointer
-3	Parameter longjmp buffer
1	Return address
2	Old base pointer
3	Function pointer
4	Longjmp buffer
5	Function pointer
6	Longjmp buffer
-2	Parameter function pointer
-1	Parameter longjmp buffer
7	Return address
8	Old base pointer
9	Function pointer
10	Longjmp buffer
11	Return address
12	Old base pointer
13	Function pointer
14	Longjmp buffer

Since each test case is meant to elucidate whether or not a particular protection mechanism halts the attack, success or failure is determined by whether the test case returns to main and prints a message (or crashes due to the protection mechanism). Failure (*i.e.*, a successful compromise) is indicated by the program dropping a shell. As a result, each test case simply returns `void`. This means that the return value modifications of vanilla error virtualization may have little to offer in this context, as no control flow decisions (indeed, no decisions at all) are made based on the return value. Error virtualization's use of rolling back the memory changes may have some effect due to restoring the program to the state it was before the routine was called and the code was injected. Therefore, we construct a simple repair policy (a snippet of which is shown in Figure 8.17) that contains an entry for each function in the testbed. Each `tp` entry tests the integrity of the stack frame and unrolls the memory changes made during the slice if this integrity fails. The amount of effort to understand the source code of the testbed (at least for the purposes of constructing the repair policy) was quite minimal: we simply extracted the function identifier and looked at how return values were handled (there were none, as each function returns `void`). As we find below, these relatively simple efforts obtain good results.

We first established two control samples for these experiments. The first control sample is set up to observe testbed behavior in the absence of all protection mechanisms. The expected result is that the attacks succeed. No protection mechanisms are used (in particular, ASLR for Fedora is turned off) and the testbed is marked as needing an executable stack with the `execstack` utility. The second control sample is based on the default behavior of Fedora Core 3; Fedora uses a form of address space layout randomization to probabilistically prevent code injection attacks. In addition, by default, it does not allow programs to execute code on the stack. We expect the control behavior of the testbed under this environment to result in crashes of the process as each attack test case is frustrated.

Compiling the Wilander testbed and then executing it for each attack form and control sample produced 36 results (out of a possible 36). Each test of the first control group produced a shell, thereby indicating a successful attack. Each attack in the ASLR control run resulted in a "Segmentation Fault" message being printed. This message presumably represents a manifestation of either the non-executable stack or ASLR in action resulting

Table 8.6: *Wilander Testbed Control Sample Results.* The test case index corresponds to the comments in Table 8.5. All attacks in the simple control group succeeded ("Shell" indicates a successful compromise). All attacks were prevented in the ASLR control group insofar as a combination of ASLR and a non-executable stack caused the program to crash.

Test Case #	**Control Result**	**ASLR Result**	**Additional Message**
-4	Shell	Segmentation Fault	
-3	Shell	Segmentation Fault	
-2	Shell	Segmentation Fault	
-1	Shell	Segmentation Fault	
1	Shell	Segmentation Fault	
2	Shell	Segmentation Fault	Attack Prevented.
3	Shell	Segmentation Fault	
4	Shell	Segmentation Fault	
5	Shell	Segmentation Fault	
6	Shell	Segmentation Fault	
7	Shell	Segmentation Fault	
8	Shell	Segmentation Fault	Attack Prevented.
9	Shell	Segmentation Fault	
10	Shell	Segmentation Fault	
11	Shell	Segmentation Fault	
12	Shell	Segmentation Fault	Attack Prevented.
13	Shell	Segmentation Fault	
14	Shell	Segmentation Fault	

from incorrect offsets used by the exploit.

In addition, a further three tests (the same in each control sample) printed a message indicating "Attack prevented." During the ASLR control sample, these three tests also ended in a segmentation fault. This phenomenon may occur for two reasons. First, while the attack may not succeed in overwriting the appropriate pointer or memory location, when it returns to `main` to print the "Attack prevented" message, the state of the stack or other important data items has been corrupted, and continuing execution (even to the bottom of `main` to simply exit) may result in a crash due to this corruption. Second, if the attack does succeed in overwriting its target critical pointer or memory location, the return to `main` may be carried out correctly, and then ASLR crashes the process. Each of these three tests has the old base pointer as a target, which supports the notion that the routine would return correctly (because the return address itself may remain undisturbed), but after the old (parent) stack frame has been restored (via the old value of `%ebp`), ASLR crashes the process. When we turned off ASLR and marked the testbed executable as needing an executable stack, every attack test case succeeded. In cases 2, 8, and 12, both the control result (no ASLR and an executable stack) and the ASLR control result print the "Attack Prevented" message. This indicates that these messages, at least for these cases, are a premature notification (*i.e.*, a bug in the testbed).

We performed two experiments to test the hypothesis that STEM can have a beneficial effect in both preventing the attacks represented by each test case as well as recovering execution. Our goal is to show that STEM can successfully prevent the attack shell as well as return safely. We execute the testbed (without ASLR in effect and with an executable stack) under STEM in two circumstances. In the first, we use a coverage policy of NONE and no repair policy. In the second, we use a simple repair policy that rolls back the changes to memory made during a routine.

When we run the testbed (more precisely, each case of the testbed) under STEM without any supervision coverage policy or repair policy, every test case results in a compromise and a shell. As in the control samples, the "Attack Prevented" messages is printed in test cases 2, 8, and 12. In accordance with our results and explanation from above, we can effectively ignore the significance of this message in these circumstances.

Table 8.7: *Wilander and STEM With and Without Repair Policy.* The test case index corresponds to the comments in Table 8.5. The apparent relatively low rate of "success" is due to the limited attack detection mechanism employed by the repair policy; this mechanism simply maintains a shadow stack and determines if the return address or saved base pointer value was overwritten. Attacks that bypass these artifacts will not be detected and thus not invoke a repair procedure. Instead, the success rate is 6 of 6 (all six attacks that STEM detected and attempted to repair were successfully repaired), rather than 6 of 18. Even if we *force* detection at the end of a routine, some of these vulnerabilities are triggered before STEM_Epilogue() executes, so repair policy will not be invoked.

Test Case #	**Without Repair Policy**	**With Repair Policy**
-4	Shell	Shell
-3	Shell	Shell
-2	Shell	Shell
-1	Shell	Shell
1	Shell	Memory Rollback, Attack Prevented
2	Shell	Memory Rollback, Attack Prevented
3	Shell	Shell
4	Shell	Shell
5	Shell	Shell
6	Shell	Shell
7	Shell	Memory Rollback, Attack Prevented
8	Shell	Memory Rollback, Attack Prevented
9	Shell	Shell
10	Shell	Shell
11	Shell	Memory Rollback, Attack Prevented
12	Shell	Memory Rollback, Attack Prevented
13	Shell	Shell
14	Shell	Shell

```
...
tp vuln_parameter_longjmp_buf &MeasureStack
  :=: &RepairStack, (unroll);

tp vuln_stack_return_addr &MeasureStack
  :=: &RepairStack, (unroll);

tp vuln_stack_base_ptr &MeasureStack
  :=: &RepairStack, (unroll);
...
```

Figure 8.17: *Sample of Wilander Repair Policy.* This policy does not work for some of the vulnerabilities in the Wilander testbed because it does not check the integrity of function pointer parameters. These vulnerabilities are ideal candidates for the use of more specialized repair policy.

When we run the testbed under a repair policy with our default intrusion sensor enabled in the repair policy, the attacks are halted and repaired in 6 of the 18 cases. The relatively low rate of success is due to the detection capabilities; we use a repair policy that employs a simple form of shadow stack integrity checking. Attacks that subvert pointers that are not checked will not kick off the remainder of the repair policy. These targets include `longjmp` buffers and function pointers that are not covered by measurements of the stack return address or saved base pointer. For all the attacks that our basic repair policy *does* detect, the policy successfully defeats the attack and allows the application to continue (although "continuing" in this case means simply returning to `main` and finishing execution). *These results match the expected results for a shadow stack detector as related in the paper describing the testbed [WK03].* We present timing information in Table 8.8 that indicates how long the actual repair process takes in these cases (an average of 25.7 milliseconds, measured via `rdtsc`).

These repairs are achieved with a relatively simple repair policy, an example snippet of which we include in Figure 8.17. This repair policy is inspired by efforts in reliability and fault tolerance that rewind execution when an error is encountered. They differ by the implied "slice off" of the functionality: the routine where an error is encountered is not "replayed" during this instance of the control flow path (in other words, it will still

Table 8.8: *Timing Information for Wilander Repairs.* The test case index corresponds to the comments in Table 8.5. The timing information is in nanoseconds and is derived from instrumentation that uses the `x86 rdtsc` instruction. All digits in the timing information are significant. As another way of estimating the work done by the repair, we provide the number of memory writes that are undone by the repair.

Test Case	**Repair Time (ns)**	**# of Restored Memory Writes**
1	13177257	53
2	39659711	65
7	16327552	61
8	60335286	81
11	35647323	57
12	38936215	77

be executed in future runs of the code). The point here is to show that even relatively simple repair policies can have a beneficial effect. Even though the actual effect may be similar to previous efforts, we can achieve it through the use of a single line of code rather than a large, specially-designed system. In these rollback and replay systems, C or C++ code (often at the OS or OS driver level) must be (re)written to change and add features and response mechanisms, requiring at least the recompilation of the replay system. With STEM, a single line of repair policy code can be added while the system is running, and the OS need not be modified.

8.2.3 Analyzing Real Vulnerabilities

In this Section, we construct repair policy for some vulnerabilities in real software, including libpng [lib], fetchmail [fet], and NULLhttpd [nul].

We note that the repair policies we discuss here necessarily seem vulnerability–specific. In particular, the IVPs trigger on vulnerability–specific conditions. The IVPs in these experiments appear artificially specific because the focus of each example is on a particular vulnerability. Of course, the general goal of automated repair would be to construct

such policies *without* foreknowledge of the vulnerability. Instead, repair policies should express constraints that we believe should hold on the behavior of the software in general. These constraints can, however, include constraints expressing well–known vulnerability conditions, and we envision the addition of standard IVPs as new vulnerability classes are discovered. We acknowledge that human foresight is not perfect, and so repair policies may be incomplete. As we will see, however, many vulnerabilities can be attributed to some type of missing sanity check on input data or data structures within a program. These types of vulnerabilities are ideal candidates for being handled with repair policy that continuously enforces sanity checking.

There seems to be a fine difference between anticipating something with a constraint expression and anticipating something with a particular exception handler. As we discuss in Chapter 5, the key functional difference is that with an exception handler, the developer has to specify arbitrary code; constraints should be much simpler to specify.

8.2.3.1 libpng

Embedding exploits in media files is a insidious form of attack. It is generally easy to trick users[2] into viewing such images by incorporating them into an email, a Web page[3], or a folder icon[4] [PMM+07]. Furthermore, users have a mistaken assumption that data content is relatively inert and thus represents less of a risk than running an arbitrary program or executable. Although this assumption is becoming less true with a trend toward active content like Flash, it has always been the case that maliciously crafted data can easily exploit vulnerabilities in the code that processes the content. The `libpng` library provides processing capabilities for the PNG (Portable Network Graphics) image format. Version 1.2.5 of this library contains several vulnerabilities [lib], including a stack-based buffer overflow due to a skipped check of a value in the image header. Over the past few years, similar code injection vulnerabilities have appeared in image processing code in both Unix and Microsoft platforms.

[2] `http://news.bbc.co.uk/2/hi/technology/6645895.stm`

[3] `http://www.theregister.co.uk/2008/01/23/booby_trapped_web_botnet_menace/`

[4] `http://www.linklogger.com/wmf_attack.htm`

```
[michael@xoren libpng]$ kview pngtest_bad.png
libpng warning: Missing PLTE before tRNS
libpng warning: Incorrect tRNS chunk length
libpng warning: tRNS: CRC error
libpng error: PNG unsigned integer out of range.
[michael@xoren libpng]$
```

Figure 8.18: *KView Using Fixed libpng.* The fixed version of `libpng` performs an extra check to output the second warning message and prevent an exploit from succeeding.

We used a sample image viewer, `rpng-x`, (akin to `display` or KView) that is provided with the PNG library. We linked `rpng-x` against the vulnerable version of `libpng`. We initiated two control runs. The first opened a regular PNG file with `rpng-x`. The file opened and displayed successfully. The second control run used the KView KDE program, which is linked against a patched version of `libpng`, to open a malformed PNG file crafted to exercise the vulnerability in `libpng`. As shown in Figure 8.18, KView survives opening the file because it uses a fixed version of `libpng`.

We then used `rpng-x` to perform two tests involving the crafted PNG file. The first test opens the file without any protection or supervision. This run, shown in Figure 8.19, terminates with a segmentation violation because the sample exploit is non-malicious; it simply overwrites the value "A" onto the stack, resulting in a crash due to a dereference of `0x41414141`. The payload could have contained a shell bound to a port.

We then executed `rpng-x` in STEM under a repair policy. Writing a repair policy for this vulnerability took a medium amount of effort. Using vanilla error virtualization does not work because the vulnerable function does not have a return value, nor does the parent function, nor does the caller of that function (the library entry point for reading a PNG image)[5]. As a result, we do not adjust the return value. We began with a simple policy that restored the stack integrity and rolled back all changes to memory made during the vulner-

[5]The `rpng-x` program invokes the reading routines from a function that does have a return value, as well as an explanation of the valid return values for that function in a source–level comment. We focus our attention on a library-specific policy, as the exploit occurs in the library, not the calling function in `rpng-x`.

```
[michael@xoren libpng]$ ./rpng-x pngtest_bad.png
libpng warning: Missing PLTE before tRNS
libpng warning: tRNS: CRC error
Segmentation fault
[michael@xoren libpng]$
```

Figure 8.19: *Exploiting libpng.* The warnings are from logic checks that are already present in `libpng`. These warnings are noted, but a logic error prevents corrective action by the library itself. Note the difference from KView's behavior in Figure 8.18. If the PNG file contained a "live" payload, this execution may have resulted in a shell.

able routine. This policy prevents the exploit from taking effect, and it continues execution of the program. The library finishes reading the header data and turns its attention to reading the image body data. However, the program crashes because of a bad dereference of a stack parameter. This stack parameter is a function pointer to a function that reads in the image data part. We thus need to adjust the repair policy to set this parameter to NULL and the function pointer to the error handler to NULL, as the program checks these conditions, logs the error with the default error handler, and terminates execution by calling `PNG_ABORT()`.

8.2.3.2 NULLhttpd

NULLhttpd contained a heap overflow vulnerability where a negative *Content-Length* header (supplied by a client, and thus untrustworthy) could cause memory overwrites by providing POST content that does not match the length of the *Content-Length* header. NULLhttpd passes the sum of the client-supplied content length plus 1024 to `calloc`. For a negative content length, this call results in a smaller request for memory from `calloc` than the POST content read routine (`ReadPOSTData`) expects. As a result, the routine can overwrite heap pointers. The patch for this vulnerability logs an error and sets the content length of the request to zero when the content length header is read.

There are three important points in execution: the place where the content length is read without a check (`read_header`), the place where memory is allocated based on this

```
symval NULL = 0;
//error_fn is offset 160 in png_struct
symval PNG_ERR_PTR_OFFSET = 0xA0;
//png_rw_ptr is at position 172 in png_struct
symval PNG_RW_PTR_OFFSET = 0xAC;

//png_ptr is first parameter (4 above return address value)
cdi png_ptr = 'ebp + 0x8;

rp SetDataWriteFunctionPtr :=:
  (mem[png_ptr + PNG_ERR_PTR_OFFSET] == NULL),
  (mem[png_ptr + PNG_RW_PTR_OFFSET] == NULL);

tp png_handle_tRNS &MeasureStack
  :=:
  &RepairStack, &SetDataWriteFunctionPtr, (unroll);
```

Figure 8.20: *Repair Policy for libpng*

```
symval ZERO = 0;
symval READ_FAILURE = -1;
symval CONTENTLENGTH = 0;
symval CONN_ADDR = 0x8051260;
symval SID_PTR_OFFSET = 0x8;

//session ID is 1st parameter
cdi sid = 'ebp + SID_PTR_OFFSET;
//the session data structure is offset in global list
cdi dat_ptr = CONN_ADDR + sid + 40;
//content length variable is a known offset in that structure
cdi in_Contentlength = dat_ptr + 16;

rp FixContentLength :=:
  (mem[in_Contentlength] == CONTENTLENGTH),
  ('rvalue == READ_FAILURE);

tp read_header
  (mem[in_Contentlength] < ZERO)
  :=:
  &FixContentLength,(unroll);
```

Figure 8.21: *Repair Policy for nullhttpd*

information (in `ReadPOSTData` — again without a sanity check), and the location where the resulting heap corruption hijacks control (a somewhat arbitrary point in execution depending on the underlying platform and memory allocation scheme). Our repair policy focuses on preventing the last location from becoming significant by detecting and fixing the first error. That is, the repair policy responds to the incorrect content length before memory is allocated and before malicious data is read in. The content length is read from the client-supplied headers in the `read_header` routine. A return value of -1 from `read_header` indicates a failure of the routine to its caller. The `read_header` function invokes `ReadPOSTData` to acquire the content of an HTTP POST request. The key challenge in constructing repair policy is to obtain a reference to the content length variable so that we can enforce a constraint on it. All serviced connections are referenced by a global variable located at address `0x8051260`. The current connection is an offset from this variable; the offset is the first argument to the `read_header` routine. The specific connection metadata is a known offset from the start of this connection descriptor structure, and the incoming content length variable is a known offset from there. Our repair policy checks if the content length is less than zero for both routines; if it is, we unroll the memory changes made in that routine, and in the case of `read_header`, assert that the return value should be -1. After an exploit attempt and the subsequent repair, NULLhttpd is able to continue running and servicing good requests.

This case is an ideal example of repair policy that does not need to get too involved and can prevent an exploit from succeeding by enforcing reasonable constraints on the data structures used by the program. In particular, the nature of this vulnerability is to cause an overwrite of heap management variables. Doing so can lead to incorrect execution in a very different place in control flow than where the actual missing checks should have been located, making it somewhat difficulty to trace the root cause of an attack. We construct a repair policy that deals with the missing checks rather than a repair policy that attempts to correct the effects of a successful exploit.

```
symval POPBUFSIZE = 513;
symval PS_PROTOCOL = 4;

tp pop3_getuidl &MeasureStack,
  :=:
  &RepairStack, ('rvalue==PS_PROTOCOL), (unroll);
```

Figure 8.22: *Repair Policy for fetchmail*

8.2.3.3 fetchmail

The `fetchmail` program is a popular and versatile mail transport agent (MTA) that can collect mail from remote servers and deliver either to local mailboxes or resend via a local MTA such as qmail or sendmail. In 2005, a buffer overflow was discovered in fetchmail's UID handling code. Carefully crafted UIDs could overflow a stack-resident fixed-size buffer and lead to remote code injection attacks. If fetchmail is running as root, this vulnerability could result in a root shell. Like some recent examples of Microsoft patches, the initial patch for this vulnerability was buggy, requiring another patch to be crafted and released.

We wrote a repair policy to monitor the `pop3_getuidl()` function and return the value `PS_PROTOCOL` (we obtained this value (4) from the fetchmail header file, where a comment indicated that it represents a protocol violation flag) if the vulnerability is tripped.

8.3 Future Work

Future work can evaluate the range of Ripple as a language. In particular, Ripple seems applicable to software assessment and testing problems that involve controlled fault injection. Fault injection is one traditional method of examining how robust programs are. Fault injection experiments, however, are difficult to set up and control for arbitrary, unmodified binary software applications. Oftentimes, the fault injection framework needs to be integrated with the software under test: a lengthy, expensive process that potentially disturbs the actual fault injection experiments. Furthermore, *fault relationships* are interesting, and it seems as if Ripple can help express the constraints on those relationships in a human-readable fashion. That is, rather than random bit flipping, we can control the *struc-*

ture of injected faults and make this capability available to the software security research community at large.

In the future, we would like to gauge how easy it is to use Ripple to inject faults into a variety of software applications in a controllable, repeatable way. We are interested in studying how effective this method of fault injection is as well as how reliable and fault–resistant these software applications are.

In less technical matters, we have not examined or evaluated the impact that ROAR has on the design process of software applications or self–healing mechanisms. This thesis is, in part, an existence proof that ROAR is useful for creating such systems. Future work can perform surveys of developers and security experts to ascertain the actual value of the workflow and identify any areas for improvement.

It would be useful to characterize the average length of a microspeculation "pipeline" for a variety of applications. Our slice and workload distribution analysis from Section 8.1.1, along with our return value predictability experiments (presented in Chapter 7) provide the basis for work in this direction. We have also created an automated test harness for evaluating error virtualization of each slice of an arbitrary software application.

We can improve STEMv2's performance by modifying the memory log implementation. It is currently based on a simple linked list that heavily exercises the underlying C library memory allocation subsystem due to frequent allocation and deallocation of slots to store the results of memory write operations. One way (an amortization–based approach) to improve performance is to preallocate memory slots based on the typical memory use of each supervised function. If we can bound the number of stores in a piece of code (*e.g.*, because STEM or another profiling tool has observed its execution), then STEM can preallocate an appropriately sized buffer. Our work on workload distribution already measures both the number and size of memory writes per slice.

8.4 Summary

Our evaluation of the work proposed in this thesis explores a number of avenues. Overall, we are interested in showing the efficacy and performance impact of our internal and external

integrity posture mechanisms and observing the workload distribution of a variety of real software. Earlier in the thesis (Chapter 3), we examined the properties of basic error virtualization to motivate some of our design choices. We find the following results:

- *The use of basic error virtualization is replete with occurrences of semantically incorrect follow-on execution.* This result motivates the need for a mechanism that provides a customizable automated error response. Nevertheless, error virtualization seems to work well for certain routines and types of programs. We have constructed a test harness that can be reused to automatically test the behavior of a wide variety of programs under error virtualization.

- *Workload distribution provides a way to inform coverage policy.* Since microspeculation can be expensive, we provide a built-in method of discovering the workload distribution (in terms of a variety of measures, including precise timing information via `rdtsc`, total memory write size, and total instructions) of a software system.

- *We have improved the performance of microspeculation supervision.* Our use of dynamic binary rewriting has drastically improved the performance of STEM over pure emulation. Furthermore, we have identified the major performance bottlenecks in the current implementation (*e.g.*, startup code, the use of the RTN API) and have a clear path of evolution for the next version to further improve performance. We can also benefit from enhancements in future versions of Pin.

- *Exploit filter generation is fast and effective.* Filter generation takes under a second and effectively blocks future attack instances. The overhead due to filter generation and enforcement — using an unoptimized prototype Java–based network proxy — is 28%.

- *Small, easy to understand repair policies can be generated.* Our work on running examples and the Wilander testbed illustrate the basics of repair policy generation. Our work with a variety of security vulnerabilities shows how to generate repair policy for problems that have occurred in practice and highlights the current limits of repair policy. Repair policy seems to provide an ideal solution for user, community, or vendor

discovered vulnerabilities. In these cases, a repair policy can quickly be developed and deployed to provide a "band aid" while a more extensive binary or source–level patch is developed and tested.

- *Repair policy interpretation occurs rapidly.* In our example programs, the Wilander testbed, and real vulnerabilities, repair takes a few tens of milliseconds. Improvements in Antlr and RealPaver can assist Ripple's performance and power.

We believe the performance impact of the system can be justified by the benefits the system provides, at least for critical applications. We may be able to rely on the use of many-core architectures, improvements in virtualization technology, or the use of Application Communities [LSK05, LSK06a] to offset the performance impact. We also anticipate moving to Pin's INS API.

Chapter 9

Discussion

This thesis does not directly address a few challenges in this space; some of these challenges represent engineering–level tasks, some are under investigation by other researchers, and some remain as open problems. Since we discuss the limitations of specific techniques in their respective chapters, this section contains a review of some of the more general limitations in this space. Many of the challenges are technical in nature, and we suggest some potential solutions. In addition, there are some issues (for example, usability or user acceptance testing) we consider out of scope of the thesis and simply do not study.

9.1 Limitations

A number of important unsolved problems remain. Specifically, we do not propose or evaluate specific methods of automatically generating repair policy. In addition, as with all systems that rely on rewinding and replaying execution [BP02, QTSZ05] after a fault has been detected, I/O with external entities remains uncontrolled. We do not provide a complete solution to this "external I/O" problem. Third, we do not provide a specific technical solution to the problem of supervising the operation of kernel code, although, conceptually, the kernel is simply another software system that can be microspeculated subject to a collection of integrity constraints. Finally, although our integrity model specifically accounts for repair validation, we do not implement solutions for validating repairs; such work is the focus of our current and future research.

9.1.1 Automatic Repair Policy Generation

This thesis does not advance a solution for automatically generating repair policy. We intend to investigate methods for doing so in future work. Specifying the solution state for constraint satisfaction remains a difficult problem; we leave as a manual exercise the task of policy specification — a task that can benefit from human expertise and is not on the critical path of runtime, real-time repair. Manual specification, however, may not scale, especially for large systems that undergo constant, widespread revisions.

An interesting research question is how to automatically determine integrity constraints on data and control flow for arbitrary software applications. We believe there are three promising avenues of research: deriving constraints from static analysis of source code, deriving constraints from symbolic execution, or learning constraints from dynamic runtime behavior. The last, although it provides efficient data collection and the most concrete and specific results, provides no guarantee of coverage: the constraints may be too narrow a slice of legal program behavior and depend heavily on the "training" or input data set. Purely static analysis, however, may generate too wide a range of valid behaviors. A process of symbolic execution that explores program paths in an offline fashion and constrains critical data items in terms of input processing seems like the most promising approach (although the computational cost seems somewhat prohibitive). Of course, the key underlying problem is that, in a completely automated system, we must trust at least one component to serve as the reference model or ground truth for program behavior. If this component is not trustworthy, the repair policy will be faulty. Fortunately, as we have pointed out, it is easy (relative to the difficulty of undoing a binary patch) to turn off a broken repair policy.

9.1.2 Speculated I/O

A supervised system may communicate with external entities that are beyond the control or logical boundary of the self–healing system. Attempts to sandbox an application's execution must sooner or later allow the application to deal with global input and output sources and sinks that are beyond the control of the sandbox. For example, if a server program supervised by STEM writes a message to a network client during microspeculation, there is no way to "take back" the message: the remote client's state has been irrevocably altered.

In our system, microspeculation can become unsafe when the speculated process slice communicates with entities beyond the control of STEM. If a transaction is not idempotent with respect to global state such as shared memory, network messages, *etc.*, then microspeculation must stop or stall before that global state is changed. Strictly speaking, the system can no longer safely speculate a code slice: the results of execution up to that point must be committed, thus limiting microspeculation's effective scope.

Repair attempts may fall short in situations where an exploit on a machine (*e.g.*, an electronic funds transfer front-end) that is being "healed" has visible effects on another machine (*e.g.*, a database that clears the actual transfer). For example, if a browser exploit initiates a PayPal transaction, even though STEM can recover control on the local machine, the user will not have an automated recourse with the PayPal system.

Such situations require additional coordination between the two systems: microspeculation must span both machines. If both machines reside in the same administrative domain, achieving this *cooperative microspeculation* is somewhat easier, but less generally applicable. While a self-healing supervision system can record I/O data, such data changes the global state of the world, and illegal or attacker-controlled messages cannot be taken back (*e.g.,* if the exploit causes financial information to be leaked, it is difficult to retrieve the data from an uncooperative or malicious communication partner interested in retaining it). A supervision system currently has no way to ask or force a peer to replay input or re-accept output — especially if that peer is malicious. Doing so requires that the protocol (and potentially the network infrastructure) support speculative messaging and entails changing the peer's implementation so that it can rewind its own execution. Since systems like STEM may not be widely deployed, we cannot rely on this type of explicit cooperation.

We could achieve a degree of cooperative microspeculation in at least four ways, each of which expresses a tradeoff between semantic correctness and invasiveness for the application, the network, and the remote communications peer.

1. **Protocol Modification** — Modify network or file system protocols and the network infrastructure to incorporate an explicit notion of speculation.

2. **Modify Communications Peer** — Modify the code of the remote entity so that it

can cooperate when the protected application is microspeculating, and thus anticipate when it may be sending or receiving a "speculated" answer or request.

3. **Gradual Commits** — Transactions can be continuously limited in scope. All memory changes occurring *before* an I/O call are marked as not undoable. Should the microspeculated slice fail, a system only undoes changes to memory made after the I/O call.

4. **Virtual Proxies** — Use buffers to record and replay I/O locally. Virtual proxies effectively serve as a man-in-the-middle during microspeculation to delay the effects of I/O on the external world. A virtual proxy serves as a delegate for a peer.

While some network and application-level protocols may already include some notion of "replay" or speculative execution, implementing widespread changes to many different protocol specifications and the network infrastructure is fairly invasive. Nevertheless, it presents an interesting technical research challenge. Another interesting possibility is to modify the execution environment or code of the remote communication partner to accept notifications from a STEM-protected application. After receiving the notification, the remote entity speculates its own I/O. While this approach promises a sound solution, it violates a requirement for transparency.

We advocate the use of a combination of virtual proxies and gradual commits because these solutions have the least impact on current application semantics and entail straightforward implementations. Since most supervision frameworks already "modify" the local entity, this approach can avoid modifying the remote entity or any protocols. Although using gradual commits and virtual proxies constrains the power of the solution, we believe it is an acceptable tradeoff, especially as self–healing systems gain traction: they should perturb legacy setups as little as possible. We note that Rx [QTSZ05] employs proxies that are somewhat akin to virtual proxies, although Rx's explicitly deal with protocol syntax and semantics during replay.

9.1.3 Kernel–Level Supervision

Since STEM operates at the user level, it cannot follow execution into the kernel; when a system call is invoked, STEM relinquishes control to the kernel, temporarily ending supervision and protection until the system call returns. This limitation is a technical one. A rough first solution to this problem would execute an entire guest OS on a VMM like PinOS or QEMU. We can then instrument and supervise both the process and the kernel with microspeculation. Note that we can still speculate only slices of the process and the kernel; we need not incur the overhead of supervision for all processes and all parts of the kernel. Another possible solution can create an infrastructure for adding a set of virtual CPUs to the kernel as loadable kernel modules; by mapping a virtual CPU to a process, we can provide microspeculation as an operating system service and follow execution into the kernel [LK06].

9.1.4 Automatic Repair Validation

System owners are understandably reluctant to permit automated changes to their environment and applications in response to attacks. The risks of doing so range from legal liability to severe consequences like death or injury (in the case of automated factory machinery or medical equipment). Although testing an automatic repair helps raise the confidence level in self-healing systems, it can never prove a correct repair.

One critical part of such testing is the verification that the changes made by the self–healing mechanism actually defeat the original attack or close variations thereof. Another problem involves ensuring that the changes or repairs do not introduce new vulnerabilities. Automatically generated fixes must be subjected to rigorous testing in an automated fashion.

This testing process defines the essence of Automatic Repair Validation (ARV), a new area of intrusion defense research. ARV consists of automatically validating each step in the ROAR workflow toward a newly healed configuration. For certain ARV problems, validating the final configuration is precisely the purpose of a solution to each problem. For example, relaunching the input responsible for the original attack against the healed configuration provides an independent sanity check on the final configuration. We distinguish between

testing the final configuration and testing individual steps (*i.e.,* detection, diagnosis, repair, deployment, *etc.*) because each of these steps may be prone to error or false positives themselves — we certainly do not wish to enact a potentially costly repair stage if it is not needed. In some cases, the complexity of the software application under consideration may preclude the construction of simple measurements of the final configuration.

ARV encompasses the entire spectrum of an automated response system's functionality: attack detection, repair accuracy, repair precision, and impact on normal behavior:

1. *Validation of detection* — The system must verify that the events causing an alert actually produce a compromise. In the case where the sensor is an anomaly detector, the detector's initial classification must be confirmed.

2. *Validation of a repair's accuracy* — The system must test and verify that the repair defeats at least the exploit input that triggered the detection. Verifying accuracy requires the identification and replay of the attack inputs. However, identifying these inputs is challenging, as they may not have been captured correctly (or at all) by the defense instrumentation. The challenge is greater if the input is contained in network traffic — data that most humans find difficult to rapidly analyze by hand.

3. *Validation of a repair's precision* — The fix must be precise, in the sense that it blocks malicious variants of the original attack and no benign input. For example, if the fix is an input filter, the system must ensure that the signature generation does not fall prey to an allergy attack [CM06].

4. *Validation of a repair's impact on application behavior* — Behavior exhibited by the application after self–healing should be similar to the previous behavior profile of the application. Researchers need to invent measures for bounding the semantic correctness or behavior of an application. Control flow graph distances may be one way to measure changes due to self-healing repairs. This ARV challenge depends greatly on quality behavior profiling.

Some repairs are better than others. For self–healing systems that perform some sort of state search, this implies that the search mechanism needs an evaluation function to rate

progress toward a certain locally or globally optimal solution. Even if exhaustive search is employed, the solution may not be sound: it may cause a semantically incorrect response. The delay, or amount of time taken to effect a repair, therefore, is probably not the best measure of the quality of the repair process.

Automatic repair validation is the subject of some of our future work and is mentioned here only for completeness. We have begun the construction of a system to deal with the second challenge listed above: identifying the precise network flows responsible for delivering an exploit. This system can take advantage of FLIPS and its content anomaly detection sensor to help manage data for replay. The exploration of ways to bound the types of errors that arise in response to changing the semantics of program execution is an open area of research. An efficient, precise profiling capability can help validate that the behavior of a healed software system is still acceptable.

9.2 Research Opportunities

Science is a process of continuous discovery. Few theories and techniques completely address all problems in an area. The work presented in this thesis enables the exploration of some interesting research directions. Deploying the supervision, repair, and exploit signature generation mechanisms we have discussed into a large community of collaborative hosts can help the community as a whole develop toward a state of protection at a low overall performance cost [Sto04, LSK06a, LSK05]. In addition, repair policy seems useful as a software engineering teaching tool. Finally, several studies on the usability of repair policy should be carried out.

9.2.1 Applicability of Repair Policy

For certain software systems, it may be difficult to identify the routines that require a repair policy. For other software, however, such a decision may be easy. Future work can study human's ability to identify routines that should be protected and to write correct policies. A second usability issue focuses on how well repair policies can be integrated with test harnesses that exercise the selected code paths and routines. Finally, how can we integrate

the use of repair policies into programming languages? Since we want to refrain from making a repair language-specific (which would reduce it to simply specifying exception handlers), investigating how hooks for policy can be embedded, perhaps through the use of Aspect-oriented Programming, seems worthy of more research.

9.2.2 Pair Programming

Pair programming is a common method of helping Computer Science students cooperatively solve problems in a small, focused group. Pair programming helps students gain communication skills, encourages sharing opinions that may not be well received by larger groups, provides a process of peer review, and allows students to rely on one another. Despite these benefits, however, pair programming can have negative side effects, especially if the exercises are too small for a two person team or if a significant disparity exists between the students' skill sets. In the former case, students may take turns doing the work. In the latter, the stronger student may shoulder the burden of the assignments out of a misplaced sense of responsibility or as an ego–boosting exercise.

Instead, it may be possible to organize pair programming teams as a purposefully adversarial (but professional) environment. The job of one of the pair is to respectfully vet and check the progress of the other. Practicing constructive criticism is a much needed skill. In this structure, one student is primarily responsible for constructing the solution to the assignment. The job of this student is to learn the source language and acquire design skills as required by the assignment. The other student acts as a "red team" or sanity check on the other student's work. They each take turns with the roles throughout the course.

The red team student can use Ripple for both fault injection and for expressing constraints to repair the computation in case their partner's code contains an error. In this way, the red team student need not focus on all the details of the assignment's source language. Their attention remains on a level of abstraction closer to the problem specification. The students can work in tandem to both generate a solution and test that solution. Their discussions to resolve the differences between the fault injection policies, repair policies, and code can help the students articulate whether such injected faults or repairs are reasonable and engage in quality peer review.

9.2.3 Application Communities

Collaborative security is a promising approach to a variety of security problems. Organizations and individuals often have a limited amount of resources to detect and respond to attacks (automated or otherwise). Allowing defenders to take advantage of the resources of their peers by sharing information related to such threats is a major step towards automating defense systems. Such an approach also presents an opportunity to alleviate the performance burden of monitoring mechanisms on participating nodes.

Collaborative security is the growing trend towards sharing information security resources within and across administrative domains and systems to improve the overall security of the peer group [Sto04]. The reasoning is that a larger and more widespread network of sensors can accumulate more accurate knowledge of an attack more quickly than a single isolated node. Collaborating peers can share exploit and vulnerability filters, raw alert streams, anonymized alerts, Self–Certifying Alerts (SCAs) [CCCR05], and other defense information.

This observation [Sto04] is shared widely in the research community. In particular, for worm detection [MS05], notification [MSVS03], and containment [AGI+03] systems, a collaborative approach is mentioned several times in the literature. A study by Moore *et al.* [MSVS03] concludes that a worm containment response needs to occur within three minutes. In addition, the participation of nearly all major ASes is required for a containment to be effective. While these requirements are challenging, the work confirms that foreseeable threats of that type are best addressed by a collaborative approach.

9.3 Musings

In time, all systems fail. As computer scientists, we use abstraction to create the illusion of a perfect world for our virtual systems. In this environment, every process has enough memory space for its code and variables. Resources are always available and execution time is plentiful. Many of the most difficult problems in computer science research originate from the failure of this fiction as it meets the constraints of the physical machines that host it and the assumptions and expectations of the users that interact with the system.

One reason that vulnerabilities exist is because specifying the "success path" of a program's state machine is already a very complex process. Thinking of ways the system can fail while specifying how it should behave is a difficult mental exercise. Other reasons include laziness, bugs in code generation or existing systems that combine with features or properties of the systems that run on top of them (*i.e.,* emergent properties), and incomplete specification, among others. In this way, IVPs seem to be specific to a class of vulnerabilities, which matches our intuition: detecting that a problem exists is still a hard task, but if reliable detectors can be generated, researchers, administrators, and users are still left with the question of how their software should respond!

Virtual systems must fail. Although we may imagine perfect systems, we lack the ability to translate the virtual ideal to a physical form while maintaining fidelity to the original. Systems contain design and implementation flaws precisely because they are the product of an inherently human process. Unintentional flaws and failures are bad enough, and such difficulties have been the focus of research in fault tolerance and dependable systems. A skilled and determined attacker compounds the problem by actively subverting a system to undermine the will, desire, and ownership rights of the system's user, operator, or administrator.

9.3.1 Ethical Considerations

> *"Shuttle reliability is uncertain, but has been estimated to range between 97 and 99 percent. If the Shuttle reliability is 98 percent, there would be a 50-50 chance of losing an Orbiter within 34 flights...The probability of maintaining at least three Orbiters in the Shuttle fleet declines to less than 50 percent after flight 113."*
>
> quote from The Office of Technology Assessment, 1989
>
> — CAIB Report, Volume 1, page 103, August 2003

Any system that proposes to take control (and therefore the ability to make a fail-stop decision) out of human hands is in need of at least a cursory discussion of liability and ethical considerations. At the very least, we may suspect that the self–healing techniques we have presented may not be best suited for medical devices, spaceflight or air traffic

control software, or nuclear reactor control software. The use of repair policies in an offline process of fault injection may still be of use for these systems. Developers and purveyors of self–healing systems should identify the extent to which their system can be relied upon (a principle codified in documents like the ACM Code of Ethics). Finally, there is an interesting moral hazard here: if self–healing succeeds, will developers become lazier and less careful?

9.3.2 Attacker Intent

Attacks themselves may be symptoms of a larger effort aimed at a nebulous (from the defender's standpoint) goal. While individual steps of an attack can be repeatedly frustrated or turned away by a self–healing mechanism, understanding the root cause or intent of an attacker's actions based on a series of attack events may help streamline repair efforts or otherwise improve the efficiency of defense. A more nimble defense can help system defenders regain the initiative from the attacker; the philosophy behind OODA decision feedback loops is an example of this capability.

This challenge is perhaps the most difficult to surmount. Security is, at its core, the imposition of one principal's will on another. Since this definition is morally neutral (it makes no judgment about the motivation of the principals), it creates the difficulty faced by any automated defense: inferring malicious intent. Determining the intent of some action or event is hard for humans, and it is unlikely that inferring intent is a concept that can be easily modeled by a computational process. Nevertheless, understanding the root cause of attacks requires making a judgment about the input and actions of components in the system. Work in this area can extend the growing field of attack graphs [SHJ+02, OBM06].

Chapter 10

Conclusion

> *It was — and still is — impossible to conduct a...full-scale simulation of the combination of loads, airflows, temperatures, pressures, vibration, and acoustics the External Tank experiences during launch and ascent...this combination of properties and composition makes foam extremely difficult to model analytically or to characterize physically...even [in] relatively static conditions, much less during the launch and ascent of the Shuttle.* **And too little effort went into understanding the origins of this variability and its failure modes.**
> — CAIB Report, Volume 1, page 52, August 2003 (emphasis added)

Many software systems are composed of multiple, complicated, and hastily devised components. These systems are deployed into noisy, complex environments and run on unreliable physical systems. Software failures are endemic to the production of computing systems. Many times, just like the physical system in the above quote, too little effort goes into understanding the failure modes of components. Software products are rushed to market or pressed into service based on unrealistic deadlines. Software developers, no matter how well educated or trained, and despite their creativity, are not trained to think about how to break code as they write it. Even if they were, simple human failings (fatigue, stubbornness, greed) allow bugs, errors, and backdoors to sneak into code.

Even organizations like NASA, with a safety-conscious culture and seemingly large amounts of resources, can fail to identify failure modes of a system (although the CAIB

Report concludes that parts of the safety culture and the amount of resources had severely degraded by early 2003). Such failures can result in the tragic loss of human life, as happened to the Columbia Shuttle on 1 February 2003[1] and the Challenger on 28 January 1986. While the root cause of both these sad and tragic incidents occurred due to hardware failures, a variety of unmanned space missions have failed or experienced a catastrophic interruption due to software errors (priority inversion, incorrect unit translation between Imperial and metric, *etc.*). Software errors have routinely interrupted flight operation systems (*e.g.*, ticketing and routing systems); a recent failure of communications systems in the Midwest[2] forced air traffic controllers to rely on tertiary systems (including personal cell phones) when radar and primary mechanisms failed.

No known method exists for removing all flaws, latent faults, and vulnerabilities from a software system before deployment. Program designs will always lack a complete description of how to handle all errors. To complicate matters, the opportunity and motivation to take advantage of these errors will not disappear as long as computing systems are trusted with the task of processing, transmitting, and storing important data. Since attacks occur at machine speed, it appears that software systems must automatically protect themselves from attacks. Such attacks may be delivered via previously unseen inputs or for previously undiscovered vulnerabilities. Furthermore, existing protection mechanisms often terminate the attacked process, thereby reducing the availability of the system.

10.1 Thesis Summary

This thesis proposed the use of machine intelligence to automatically strengthen both the internal and external *integrity postures* of a software system. Preserving the integrity of a system can assist in preserving the availability of the system. Specifically, we proposed speculatively executing a software system subject to a *repair policy* for memory corruption vulnerabilities. We employ constraint satisfaction driven by the content of the repair pol-

[1] Columbia, designated STS-107, was actually the 113^{th} Shuttle mission, which makes the OTA quote in the previous Chapter even more prescient.

[2] `http://www.reuters.com/article/domesticNews/idUSN2541697920070925?sp=true`

icy to automatically repair the integrity of critical data items. In addition, we proposed automatically generating exploit signatures (most existing commercial malware signatures are manually generated or vetted) to insulate the input boundary of the system from the reintroduction of confirmed malicious input. Such a dual–pronged approach provides redundancy of protection; it simultaneously provides both a healing function and a prophylactic measure.

We designed and constructed three key systems: STEM, FLIPS, and Ripple. STEM provides a microspeculated runtime environment. FLIPS automatically generates exploit signatures based on the classifications of a content anomaly sensor and feedback from STEM. Ripple is our repair policy language based on our extensions to the Clark-Wilson Integrity Model. We combine these components into a novel architecture that implements our proposed self–healing workflow, ROAR, and performs lightweight binary supervision of binary–only and source–available software systems.

One of our main assumptions is that specifying the boolean conditions of a repair policy is less complex and less error-prone than specifying arbitrary repair code in exception handlers for unanticipated errors. These conditions serve as a goal state for repair, and we advocate letting the system automatically find the best value assignments to achieve the goal state. This process relieves the programmer of the burden of prescience: he need not fix all unknown vulnerabilities ahead of time. It is enough to express what properties the code *should* maintain. Programmers have access to a developing codebase. The users of the system have access to the system as a whole, and should find it easier to write repair policy.

10.2 Results Summary

The technical goal of this thesis is to provide an environment where both supervision and automatic remediation can take place to support self–healing software. Recall our thesis statement from Chapter 1:

Thesis Statement: *This dissertation examines the claim that it is possible to speculatively execute a software system subject to an integrity repair policy so that faults and exercised vulnerabilities are automatically remedied by machine intelligence to preserve both the internal and external*

integrity postures of a software system. In addition, microspeculation–based supervision can help generate exploit signatures to protect the system input boundary.

In order to support this hypothesis, we conducted a number of experiments as detailed in Chapter 3, Chapter 6, Chapter 7, and Chapter 8. These experiments were aimed at showing a variety of properties of our self–healing and repair techniques and how they apply to memory corruption vulnerabilities. We have learned the following lessons:

- Error virtualization must be evaluated on an application by application basis. For the software we examined, which was not the target software class for error virtualization, it works from 3% to 60% of the time. More experimentation is needed to discover the rate at which it degrades if multiple functions undergo error virtualization, as well as how well it applies to other software classes.

- Repair policy can support customizable error virtualization, especially for common library routines with standard failure codes.

- Repair policy can fill a much needed role: a specification to control the behavior of vulnerable applications without the need for a source–level or binary patch. In particular, vendors and security professionals can write and distribute repair policy for vulnerabilities that they discover before a binary patch is fully tested and available.

- Repair policy helps heal cases in the Wilander testbed. We show how STEM and repair policy helps heal both examples of semantically incorrect continuation of execution as well as vulnerabilities in real software.

- Good detection capabilities are still important. Although repair policy can be triggered by measurements of important Constrained Data Items (CDI), oftentimes it is difficult to predict which CDIs will be corrupted in any given attack, and should therefore become the subject of an IVP. In these cases, Ripple and STEM can rely on general detection mechanisms to trigger the repair. Thus, repair policy can be somewhat agnostic to what triggers a repair while still providing a mechanism for detection to supplement any attack sensors in place in STEM.

- During our analysis of real vulnerabilities, it became clear that Ripple requires a much better type system so that it can interact with data structures that live on the heap. The current Ripple system only deals reliably with data that resides at fixed addresses (addresses known before runtime) or addresses relative to a stack variable.

- Repairs occur in a few tens of milliseconds. For the Wilander testbed, repairs proceed in about 25 milliseconds. Repair for our two example vulnerabilities takes about the same time. Repair for the `libpng` vulnerability takes about 52 milliseconds. The `NULLhttpd` vulnerability and the `fetchmail` vulnerability both take similar amounts of time.

- STEMv1 imposes roughly a 30X performance cost on whole system execution due to unoptimized emulation (*e.g.*, no translated instruction cache). This performance cost can be significantly reduced by only supervising small parts of an application. With the addition of optimizations like an instruction cache, we can expect to achieve more significant speedups, similar to the experience of the RIO system [BGA03].

- STEMv2 helps discover the workload distribution of individual software applications to assist development of reasonable coverage policies.

- STEMv2 imposes a 1X to 2X performance cost on what we term the "mainline" execution of a program: the portion of a program that executes after startup and dynamic library loading.

- An implementation choice causes STEMv2 to impose a performance penalty on system startup. This penalty is especially apparent in programs that utilize large amounts of GUI code or in programs that are short-lived. This penalty exists due to the use of Pin's "RTN" API instrumentation. It appears that, as an object is loaded, every routine is instrumented, even routines that may not otherwise be encountered in execution. In addition, our implementation associates some instructions with multiple analysis routines to make development complexity manageable and STEM's source code understandable.

- Automatic exploit filter generation occurs quickly and is useful for protecting the

input boundary of a system in conjunction with strengthening the internal integrity posture of a system.

10.3 Closing

This thesis recounts my research on software systems self–defense mechanisms that seek to automatically remedy the effects of a fault or vulnerability so that execution continues safely and maintains system availability. Relative to existing work, this thesis has four main novel contributions: the ROAR workflow, selective policy–constrained speculative execution, repair policy, and automatic exploit signature generation. No system provides perfect security, but we can provide well–formed recovery mechanisms and automatically invoke them. An integrity repair model assists in bridging the gap between current systems and systems that can automatically self–heal.

Appendix A

Ripple

This Appendix provides a brief overview for the Ripple language.

A.1 Ripple Language Tutorial

At its core, a Ripple program enumerates mappings between lists of boolean constraints and functions (referred to as transformation procedures or TPs). TPs are declared using the `tp` keyword followed by the name or callsite address of the function or routine that Ripple supervises. The execution of a TP is bound to a set of repair constraints with the binding operator `:=:` symbol.

The simplest thing for Ripple to do is supervise a routine. If the execution supervision environment detects a problem during the routine, then we can overwrite the return value of the routine with an error code. The special variable `'rvalue` represents the return value; Ripple automatically maps it to the appropriate underlying representation (*e.g.*, the %eax register for x86 programs). The apostrophe (`'`) symbol indicates that `rvalue` corresponds to the built-in variable — not a user-specified CDI variable named "rvalue". The semi-colon is a statement terminator, as in languages like C and Java.

```
tp hello :=:
  ('rvalue == 3780);
```

Note that the "repair" actions implied by the constraint are *always* executed **if** the

supervision environment detects an error or fault occurring during the execution of the `hello` routine. We can force these constraints to be evaluated regardless of what the supervision environment believes using the `true` keyword.

```
tp hello true :=:
  ('rvalue == 3780);
```

We can also force the constraint to never be enforced (although this is rarely useful) using the `false` keyword. One use of this capability may be to ship a default policy with all supervision disabled. As the program runs, we can automatically & selectively edit the file and reload the policy in response to some external feedback, but that technique is beyond the scope of this tutorial.

```
tp hello false :=:
  ('rvalue == 3780);
```

This tutorial will shortly consider how to limit the invocation of the constraint repair with measurements on data items rather than the three basic techniques we have just seen (*i.e.*, deferring to the supervision environment: the default; using the `true` keyword to always repair; and using the `false` keyword to never repair).

It is critical to note that the constraints that are bound to a TP do **not** specify *how* to accomplish something; rather, they only specify *what should be true* after the repair of the TP completes. The interpreter is free to perform whatever changes it thinks are necessary to achieve the end goal and satisfy the constraint. Thus, in this example, our code is not instructing Ripple to assign the value 3780 into the variable `'rvalue`: instead, it specifies that `'rvalue` **must** hold the value 3780 after the repair completes. The Ripple interpreter is free to decide exactly how such a goal may be accomplished.

We can augment this simple example with an "action" that reverses the changes to memory during the supervised procedure. The `unroll` keyword is a special type of constraint: it is not a variable that Ripple compares with anything; instead, it causes the supervision environment to undo all changes to memory made during the execution of the associated TP (in this case, `hello`).

```
tp hello :=:
  (unroll),
  ('rvalue == 3780);
```

Of course, hard-coded numbers are a Bad Thing (TM). Ripple provides a way to declare symbols that are seen only within the context of the language itself: the `symval` keyword. The syntax of this keyword is very similar to a variable declaration in C, while the semantics of the keyword are similar to the `#define` or use of the `const` keyword in C: the declared symbol is a constant. The following program defines a symbol named "FAIL" whose value is set to 3780. The value of FAIL cannot be altered after this point, and the parser ignores later re-declarations.

```
symval FAIL = 3780;
tp hello :=:
  (unroll),
  ('rvalue == FAIL);
```

The core data type in a Ripple program is a Constrained Data Item (CDI). A CDI can be associated with a set of boolean relations that constrain the range of its value. We can declare a CDI in Ripple by using the `cdi` keyword. While CDIs may seem like source-level variables (and one common use case treats them as such), CDIs can be mapped to a number of constructs via hardware artifacts. In fact, since Ripple is interpreted by a binary supervision environment, it is most natural to map a CDI to a certain portion of primary memory. A CDI must be declared and mapped in the same Ripple statement. For example, the following statement declares a CDI named `cansayhello` that uses the mapping operator `=>` to associate `cansayhello` with the byte of primary memory at virtual address 0xbf87a3d4. Future uses of the variable name `cansayhello` refer to that memory byte. The keyword `mem` refers to the memory at address zero. The operators '[' and ']' are used to indicate an offset from `mem`, much like array access in languages like C and Java. One cannot operate on a CDI without declaring and mapping it to a physical or virtual entity that is managed by the supervision environment.

```
cdi cansayhello => mem[0xbf87a3d4];
```

As alluded to earlier, Ripple programs can rely on external detection capabilities transparently provided by the supervision environment in order to invoke the enforcement of the constraints associated with a particular TP. However, Ripple also provides the ability to explicitly define "detection" or measurement points. Such measurement points are called Integrity Verification Procedures or IVPs. An IVP is similar in syntax to a TP, except that it resolves to a boolean TRUE or FALSE value by evaluating the conjunction of all the boolean conditions on CDIs bound to the IVP name. An IVP is declared using the `ivp` keyword and supplying the name of the IVP, followed by the binding operator and a list of boolean conditions on CDIs, much like a TP is declared. One major difference is that one cannot bind an "action" like `unroll` to an IVP.

```
cdi cansayhello => mem[0xbf87a3d4];


ivp MeasureHello :=:
   (cansayhello == true);
```

The previous code declares a CDI named `cansayhello` and maps it to a particular virtual memory address. It then declares an IVP named `MeasureHello` and binds one assertion to it. The assertion states that the CDI named `cansayhello` (effectively, the byte at memory address 0xbf87a3d4) should be equivalent to the value of the Ripple keyword `true` when the IVP is invoked. The value of the keyword `true` is non-zero; the value of the Ripple keyword `false` is zero. These declarations create a procedure for measuring the integrity of a particular CDI, but we need to inform the supervision system to evaluate the IVP at a particular point in execution.

The next program combines a number of techniques we have previously examined to declare a CDI, IVP, and TP and provide a response by restoring the integrity of another CDI (the built-in variable `'rvalue`). Binding an IVP to a TP is accomplished by using the '&' operator followed by the IVP name when declaring a TP.

```
symval FAIL_WITH_ERRORX = 3780;
symval ADDRESSOF_CANSAY = 0xbf87a3d4;
```

```
cdi cansayhello => mem[ADDRESSOF_CANSAY];

ivp MeasureHello :=:
   (cansayhello == true);

tp hello &MeasureHello :=:
  (unroll),
  ('rvalue == FAIL_WITH_ERRORX);
```

The semantics of the above Ripple program are straightforward. In effect, this program means: "*After the function* `hello` *has executed, compare the value of memory address 0xbf87a3d4 with the value 0. If they are the same, then do not invoke the RP epilogue for* `hello`*. If they are not the same, then the IVP has failed, and the integrity of the CDI* `cansayhello` *is no longer sound. In this case, the system must effect a self–healing response. This response involves the special action of rolling back all memory changes made during the TP* `hello` *as well as setting the return value of* `hello` *to 3780.*"

Measuring source-level CDIs during an IVP is useful, but often attacks actually subvert data structures and items that are not visible to the application itself, such as the return address on the program stack. Ripple aids in detection by exposing access to primitives that are useful in measuring the integrity of these "hidden" CDIs.

The following example illustrates the definition of an IVP that uses two special built-in variables: `'raddress` and `'shadowstack`. The IVP makes sure that the return address that is currently in the appropriate register matches the stack that STEM has been keeping track of. The example also shows that `'shadowstack` is a construct that is similar to `'mem` — a programmer can combine a reference to it with array-style access by using the bracket operators.

```
ivp MeasureStack :=:
   ('raddress == 'shadowstack[0]);

tp hello &MeasureStack :=:
```

```
  (unroll),
  ('rvalue == 3780);
```

This code example defines an IVP called `MeasureStack` that binds a constraint involving the `'raddress` CDI and the first (*i.e.*, top) position of STEM's shadow return address stack. This IVP is then bound to the TP `hello` so that the relation between `'raddress` and `'shadowstack[0]` is measured immediately before `hello` returns. If the relation fails (*i.e.*, the contents of these variables do not match, then the RP of `hello` runs and ensures that the memory writes made during `hello` are undone and the built-in CDI `'rvalue` (the return value of `hello`) is set to 3780. Naturally, we could use the `symval` keyword to provide a symbolic name for the return value.

An IVP can contain multiple constraints. For example, the IVP `MeasureHello` contains two clauses, one involving a CDI defined by the Ripple policy author and the other involving the built-in CDIs `'raddress` and `'shadowstack`. As in the constraint listing for a TP, the comma operator indicates the conjunction of the basic relation atoms. Note also the use of the `symval` keyword to define an offset of zero for the "top" of the shadow return address stack.

```
symval FAIL_WITH_ERRORX = 3780;
symval ADDRESSOF_CANSAY = 0xbf87a3d4;
symval TOP = 0;

cdi cansayhello => mem[ADDRESSOF_CANSAY];

ivp MeasureHello :=:
   (cansayhello == true),
   ('raddress == 'shadowstack[TOP]);

tp hello &MeasureHello :=:
  (unroll),
  ('rvalue == FAIL_WITH_ERRORX);
```

Finally, we may desire to check multiple IVPs at the conclusion of a particular TP. While one IVP could contain a number of constraint clauses, for modularity and ease of reference, it may be desirable to associate various constraints under a particular logical grouping. Ripple provides the ability to invoke groups of these constraints by binding multiple IVPs to a TP. In order to associate a number of IVPs with a TP, a Ripple programmer uses the '&' and comma operators as in the following example. The comma operator syntax naturally mirrors its similar use in defining the conjunction of constraints bound to an IVP or TP. Here, it indicates that *all* IVPs should successfully return a value of `true` in order for the TP to complete *without* invoking the associated repairs.

```
symval FAIL_WITH_ERRORX = 3780;
symval ADDRESSOF_CANSAY = 0xbf87a3d4;
symval TOP = 0;

cdi cansayhello => mem[ADDRESSOF_CANSAY];

ivp MeasureStack :=:
   ('raddress == 'shadowstack[TOP]);

ivp MeasureHello :=:
   (cansayhello == true);

tp hello &MeasureStack, &MeasureHello :=:
  (unroll),
  ('rvalue == FAIL_WITH_ERRORX);
```

Sometimes, the amount of work (*i.e,* the number of constraints) that a TP has to evaluate becomes quite large. Ripple provides a construct called a Repair Procedure (RP) that can group a set of constraints that the Ripple interpreter should invoke when a TP needs to be repaired. RPs are defined using the `rp` keyword. The `unroll` keyword is in fact a built-in RP. In this way, a TP can simply list the RPs that must be invoked during

healing rather than a long list of complex constraint expressions. Moreover, RPs provide a Ripple programmer with the ability to gather and define repair constraints in one program text location rather than duplicating such lists throughout the policy text with a potentially large number of TPs.

```
symval FAIL_WITH_ERRORX = 3780;
symval ADDRESSOF_CANSAY = 0xbf87a3d4;
symval TOP = 0;

cdi cansayhello => mem[ADDRESSOF_CANSAY];

ivp MeasureHello :=:
   (cansayhello == true);

ivp MeasureStack :=:
   ('raddress == 'shadowstack[TOP]);

rp FixHello :=:
   ('rvalue == FAIL_WITH_ERRORX);

tp hello
   &MeasureStack,
   &MeasureHello
   :=:
   (unroll),
   &FixHello;
```

The basic techniques presented so far in this tutorial can be combined to construct simple but powerful constraint interactions that govern the execution integrity of many applications. In particular, various definitions and combinations of IVPs with constraint listings (RPs) that are bound to that TP provide a comprehensive way to govern the

correctness of a piece of software as it attempts to recover from code injection attacks, among other exploits. However, there are many advanced features of the language — especially dealing with mapping to source-level data constructs and types as well as dynamically defining and binding a Repair Procedure (RP) to a TP over time.

A.2 Ripple One-Liners

Ripple can accomplish a lot using just a little code. Here are some examples of one-line Ripple programs that can have a big impact, illustrate a useful feature of the language, and provide a starting point for more interesting combinations.

```
tp 'any true :=: ('rvalue==0);
```

This Ripple program forces the return value of any TP that is executed, regardless of whether a fault occurs, to be zero.

```
tp 'any true :=: ('rvalue==-1);
```

A similar program can force the return value of any TP to be negative one.

```
tp 'any true :=: (unroll);
```

This Ripple program forces all functions to undo the changes made to memory during their execution. A useful fault injection tool.

```
tp 'any :=: (unroll), ('rvalue==-1);
```

This Ripple program combines the previous two examples, but allows the built-in IVP monitors to determine if the repairs represented by `unroll` and an `rvalue` of negative one should be enacted. This program is, for the most part, equivalent to vanilla error virtualization.

```
tp 'any true :=: (mem[0x0..'memend]==0);
```

Ripple can force all the memory of a process to hold the value `0x0`. Another useful fault injection tool.

```
tp main true :=: ('stdout=="Hello, world.");
```

Of course, no language tutorial is complete without a "Hello, world." example. Ripple's "Hello, world." takes just one line. The semantics of setting the special CDI `'stdout` to a string value is to print that value to the supervised program's standard output stream. Setting it to a boolean value will print a string representation of that value. Setting it to a numeric value will print a string representation of that value. Setting it equal to a known CDI name will print the value of that CDI.

```
tp 'any true :=: ('stdout=="Hello, world.");
```

An alternative "Hello, world." where the message is printed for every function.

A.3 Planned Improvements

In its current form, Ripple provides a useful tool. But we are planning several major improvements, including dynamically binding a particular set of RPs to a TP, and dealing with return values that hold addresses of functions or data structures.

We can decide which RP to invoke at runtime (rather than specifically saying so in the policy itself prior to runtime) by selecting from a list of RPs based on some conditional test (very similar to the ternary operation in languages like C, Java, and C++). For example, the following program selects between two possible RPs given a test of the CDI x. Note how references to RPs and direct invocation of asserts can peacefully coexist.

```
tp my_foo &MyFooIVP ?=? (x<5) <- ('rvalue=10) : &YourRP;
```

Furthermore, we can provide a partly dynamic definition of an RP based on runtime conditions.

```
rp MyRP (x<5) :=: ('rvalue==5) : ('rvalue==4);
```

Another interesting challenge is to support constraints that are existential; that is, they express that a certain data type or structure is not NULL, but rather points to a valid instance of that type. Effectively, `'rvalue` would need to be set to the address of such an

object instance. There are two challenges here: the first involves identifying an appropriate existing object on the memory heap and the second involves having the WVM set up such an object by itself. To do so, we need to refine Ripple's ability to express types, provide it with the ability to allocate new memory in the process's address space, express constraints about appropriate initial default values for this type, and obtain the pointer to this new memory.

Bibliography

[ABEL05] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2005.

[AGI+03] Kostas Anagnostakis, Michael B. Greenwald, Sotiris Ioannidis, Angelos D. Keromytis, and Dekai Li. A Cooperative Immunization System for an Untrusting Internet. In *Proceedings of the* 11^{th} *IEEE International Conference on Networks (ICON)*, pages 403–408, October 2003.

[ASA+05] Kostas G. Anagnostakis, Stelios Sidiroglou, Periklis Akritidis, Konstantinos Xinidis, Evangelos Markatos, and Angelos D. Keromytis. Detecting Targeted Attacks Using Shadow Honeypots. In *Proceedings of the* 14^{th} *USENIX Security Symposium.*, pages 129–144, August 2005.

[ati03] Adaptive Use of Network-Centric Mechanisms in Cyber-Defense. In *Proceedings of the* 2^{nd} *IEEE International Symposium on Network Computing and Applications*, April 2003.

[Avi85] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, 11(12):1491–1501, 1985.

[BAF+03] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized Instruction Set Emulation to Distrupt Binary Code Injection Attacks. In *Proceedings of the* 10^{th} *ACM Conference on Computer and Communications Security (CCS)*, October 2003.

[BBW+07] Nikita Borisov, David J. Brumley, Helen J. Wang, John Dunagan, Pallavi Joshi, and Chuanxiong Guo. A Generic Application-Level Protocol Analyzer and its Language. In *Proceedings of the* 14^{th} *Symposium on Network & Distributed System Security (NDSS)*, Feb 2007.

[BCS06] Sandeep Bhatkar, Abhishek Chaturvedi, and R. Sekar. Improving Attack Detection in Host-Based IDS by Learning Properties of System Call Arguments. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2006.

[BDS03] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the* 12^{th} *USENIX Security Symposium*, pages 105–120, August 2003.

[Bel05] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the 2005 USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, April 2005.

[BGA03] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 265–275, 2003.

[BK04] Stephen Boyd and Angelos Keromytis. SQLrand: Preventing SQL Injection Attacks. In *Proceedings of the* 2^{nd} *Applied Cryptography and Network Security (ACNS)*, pages 292–302, June 2004.

[BKL90] S. Brilliant, J. C. Knight, and N. G. Leveson. Analysis of Faults in an N-Version Software Experiment. *IEEE Transactions on Software Engineering*, 16(2), February 1990.

[BNS+06] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards Automatic Generation of Vulnerability-Based Signatures. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2006.

[BOH02] J. Bowring, A. Orso, and M. J. Harrold. Monitoring Deployed Software Using Software Tomography. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, November 2002.

[BP02] A. Brown and D. A. Patterson. Rewind, Repair, Replay: Three R's to dependability. In 10^{th} *ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

[BST00] A. Baratloo, Navjot Singh, and Timothy Tsai. Transparent Run-Time Defense Against Stack Smashing Attacks. In *Proceedings of the USENIX Annual Technical Conference*, June 2000.

[CB05] Ramkumar Chinchani and Eric Van Den Berg. A Fast Static Analysis Approach to Detect Exploit Code Inside Network Flows. In *Proceedings of the* 8^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 284–304, September 2005.

[CC02] Suresh N. Chari and Pau-Chen Cheng. BlueBoX: A Policy–driven, Host–Based Intrusion Detection System. In *Proceedings of the* 9^{th} *Symposium on Network and Distributed Systems Security (NDSS 2002)*, 2002.

[CCCR05] Manuel Costa, Jon Crowcroft, Miguel Castro, and Antony Rowstron. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP)*, 2005.

[CCZ+07] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. Bouncer: Securing Software By Blocking Bad Input. In *Proceedings of the ACM Symposium on Systems and Operating Systems Principles (SOSP)*, 2007.

[CF03] George Candea and Armando Fox. Crash-Only Software. In *Proceedings of the* 9^{th} *Workshop on Hot Topics in Operating Systems (HOTOS-IX)*, May 2003.

[CGP+06] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: A System for Automatically Generating Inputs of Death

Using Symbolic Execution. In *Proceedings of the* 13^{th} *ACM Conference on Computer and Communications Security (CCS)*, 2006.

[CM02] Frederic Cuppens and Alexandre Miege. Alert Correlation in a Cooperative Intrusion Detection Framework. In *IEEE Security and Privacy*, 2002.

[CM06] Simon P. Chung and Aloysius K. Mok. Allergy Attack Against Automatic Signature Generation. In *Proceedings of the* 9^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2006.

[CPM+98] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the USENIX Security Symposium*, 1998.

[CPWL07] Weidong Cui, Marcus Peinado, Helen J. Wang, and Michael E. Locasto. ShieldGen: Automated Data Patch Generation for Unknown Vulnerabilities with Informed Probing. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2007.

[CS02] M. Chew and D. Song. Mitigating Buffer Overflows by Operating System Randomization. Technical Report CMU-CS-02-197, Carnegie Mellon University, December 2002.

[CSL+08] Gabriela F. Cretu, Angelos Stavrou, Michael E. Locasto, Angelos D. Keromytis, and Salvatore J. Stolfo. Casting Out Demons: Sanitizing Training Data for Anomaly Sensors. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.

[CSSK07] Gabriela F. Cretu, Angelos Stavrou, Salvatore J. Stolfo, and Angelos D. Keromytis. Data Sanitization: Improving the Forensic Utility of Anomaly Detection Systems. In *Workshop on Hot Topics in System Dependability (HotDep)*, pages 64–70, June 2007.

[CSWC05] Jedidiah R. Crandall, Zhendong Su, S. Felix Wu, and Frederic T. Chong. On Deriving Unknown Vulnerabilities from Zero-Day Polymorphic and Metamorphic Worm Exploits. In *Proceedings of the* 12^{th} *ACM Conference on Computer and Communications Security (CCS)*, November 2005.

[CW87] David D. Clark and David R. Wilson. A Comparison of Commercial and Military Computer Security Policies. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1987.

[CXS+05] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the* 14^{th} *USENIX Security Symposium*, pages 177–191, August 2005.

[DA02] E. Duesterwald and S. P. Amarsinghe. On the Run – Building Dynamic Program Modifiers for Optimization, Introspection, and Security. In *Conference on Programming Language Design and Implementation (PLDI)*, 2002.

[DKC+02] G. W. Dunlap, S. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, February 2002.

[DR03] Brian Demsky and Martin C. Rinard. Automatic Detection and Repair of Errors in Data Structures. In *Proceedings of the* 18^{th} *Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, October 2003.

[EFB01] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, 2001.

[EPP98] Marius Evers, Sanjay J. Patel, and Yale N. Patt. An Analysis of Correlation and Predictability: What Makes Two-Level Branch Predictors Work. In *Proceedings of the* 25^{th} *International Symposium on Computer Architecture*, June 1998.

[Eto00] J. Etoh. GCC Extension for Protecting Applications From Stack-smashing Attacks. In *http://www.trl.ibm.com/projects/security/ssp*, June 2000.

[FBF99] Timothy Fraser, Lee Badger, and Mark Feldman. Hardening COTS Software with Generic Software Wrappers. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, 1999.

[fet] `http://fetchmail.berlios.de/fetchmail-SA-2005-01.txt`.

[FKF+03] Henry H. Feng, Oleg Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly Detection Using Call Stack Information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, May 2003.

[FL06] Prahlad Fogla and Wenke Lee. Evading Network Anomaly Detection Systems: Formal Reasoning and Practical Techniques. In *Proceedings of the* 13^{th} *ACM Conference on Computer and Communications Security (CCS)*, pages 59–68, 2006.

[FSA97] S. Forrest, A. Somayaji, and D. Ackley. Building Diverse Computer Systems. In *Proceedings of the* 6^{th} *Workshop on Hot Topics in Operating Systems*, pages 67–72, 1997.

[GDJ+05] Jonathon T. Giffin, David Dagon, Somesh Jha, Wenke Lee, and Barton P. Miller. Environment-Sensitive Intrusion Detection. In *Proceedings of the* 8^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2005.

[Gee03] D. E. Geer. Monopoly Considered Harmful. *IEEE Security & Privacy*, 1(6):14 & 17, November/December 2003.

[GJJ06] Vinod Ganapathy, Trent Jaeger, and Somesh Jha. Retrofitting Legacy Code for Authorization Policy Enforcement. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2006.

[Got03] G. Goth. Addressing the Monoculture. *IEEE Security & Privacy*, 1(6):8–10, November/December 2003.

[GR03] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In 10^{th} *ISOC Symposium on Network and Distributed Systems Security (SNDSS)*, February 2003.

[Gra04] Laurent Granvilliers. RealPaver User's Manual. RealPaver User Manual, 2004.

[GRS04] Debin Gao, Michael K. Reiter, and Dawn Song. Gray-Box Extraction of Execution Graphs for Anomaly Detection. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2004.

[GRS05] Debin Gao, Michael K. Reiter, and Dawn Song. Behavioral Distance for Intrusion Detection. In *Proceedings of the* 8^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 63–81, September 2005.

[GV99] Anup K. Ghosh and Jeffery M. Voas. Innoculating Software for Survivability. *Communications of the ACM*, 42(7), 1999.

[HLS04] David A. Holland, Ada T. Lim, and Margo I. Seltzer. An Architecture a Day Keeps The Hacker Away. In *Proceedings of the Workshop on Architectural Support for Security and Anti-Virus (WASSA)*, October 2004.

[HP03] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 3^{rd} edition, 2003.

[HPK01] Mark Handley, Vern Paxson, and Christian Kreibich. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In *Proceedings of the USENIX Security Conference*, 2001.

[HSF98] S. A. Hofmeyr, Anil Somayaji, and S. Forrest. Intrusion Detection System Using Sequences of System Calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[IKBS00] Sotiris Ioannidis, Angelos D. Keromytis, Steven M. Bellovin, and Jonathan M. Smith. Implementing a Distributed Firewall. In *Proceedings of the* 7^{th} *ACM International Conference on Computer and Communications Security (CCS)*, pages 190–199, November 2000.

[JKDC05] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting Past and Present Intrusions through Vulnerability-Specific Predicates. In *Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP)*, 2005.

[KBA02] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of the* 11^{th} *USENIX Security Symposium*, August 2002.

[KBO+05] Benjamin A. Kuperman, Carla E. Brodley, Hilmi Ozdoganoglu, T. N. Vijaykumar, and Ankit Jalote. Detection and Prevention of Stack Buffer Overflow Attacks. *Communications of the ACM*, 48(11):51–56, November 2005.

[KC03] Samuel T. King and Peter M. Chen. Backtracking Intrusions. In 19^{th} *ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.

[KF02] O. Patrick Kreidl and Tiffany M. Frazier. Feedback Control Applied to Survivability: A Host-Based Autonomic Defense System. *IEEE Transactions on Reliability*, 2002.

[KK04] Hyang-Ah Kim and Brad Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *Proceedings of the USENIX Security Conference*, 2004.

[KKM+05] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Polymorphic Worm Detection Using Structural Information of Executables. In *Proceedings of the* 8^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 207–226, September 2005.

[KKP03] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proceedings of the* 10^{th} *ACM Conference on Computer and Communications Security (CCS)*, pages 272–280, October 2003.

[KMLC05] Samuel T. King, Z. Morley Mao, Dominic G. Lucchetti, and Peter M. Chen. Enriching Intrusion Alerts Through Multi-host Causality. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS)*, 2005.

[KPG+03] Angelos D. Keromytis, Janak Parekh, Philip N. Gross, Gail Kaiser, Vishal Misra, Jason Nieh, Dan Rubenstein, and Sal Stolfo. A Holistic Approach to Service Survivability. In *Proceedings of the* 1^{st} *ACM Workshop on Survivable and Self-Regenerative Systems (SSRS)*, pages 11–22, October 2003.

[KTK02] Christopher Kruegel, Thomas Toth, and Engin Kirda. Service Specific Anomaly Detection for Network Intrusion Detection. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, 2002.

[LAZJ03] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug Isolation via Remote Program Sampling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[LcC04] Lap Chung Lam and Tzi cker Chiueh. Automatic Extraction of Accurate Application-Specific Sandboxing Policy. In *Proceedings of the* 7^{th} *International Symposium on Recent Advances in Intrusion Detection*, September 2004.

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of Programming Language Design and Implementation (PLDI)*, June 2005.

[lib] `http://www.us-cert.gov/cas/techalerts/TA04-217A.html`.

[LK06] Michael E. Locasto and Angelos D. Keromytis. Speculative Execution as an Operating System Service. Technical Report CUCS-024-06, Columbia University, 2006.

[LKMS03] Ruby B. Lee, David K. Karig, John P. McGregor, and Zhijie Shi. Enlisting Hardware Architecture to Thwart Malicious Code Injection. In *Proceedings of*

the International Conference on Security in Pervasive Computing (SPC-2003), Lecture Notes in Computer Science, Springer Verlag, March 2003.

[LNZ+] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable Statistical Bug Isolation. In *PLDI 2005*.

[LPKS05] Michael E. Locasto, Janak J. Parekh, Angelos D. Keromytis, and Salvatore J. Stolfo. Towards Collaborative Security and P2P Intrusion Detection. In *Proceedings of the IEEE Information Assurance Workshop (IAW)*, pages 333–339, June 2005.

[LS05] Zhenkai Liang and R. Sekar. Fast and Automated Generation of Attack Signatures: A Basis for Building Self-Protecting Servers. In *Proceedings of the* 12^{th} *ACM Conference on Computer and Communications Security (CCS)*, November 2005.

[LSCK07] Michael E. Locasto, Angelos Stavrou, Gabriela F. Cretu, and Angelos D. Keromytis. From STEM to SEAD: Speculative Execution for Automatic Defense. In *Proceedings of the USENIX Annual Technical Conference*, pages 219–232, June 2007.

[LSK05] Michael E. Locasto, Stelios Sidiroglou, and Angelos D. Keromytis. Application Communities: Using Monoculture for Dependability. In *Proceedings of the* 1^{st} *Workshop on Hot Topics in System Dependability (HotDep-05)*, pages 288–292, June 2005.

[LSK06a] Michael E. Locasto, Stelios Sidiroglou, and Angelos D. Keromytis. Software Self-Healing Using Collaborative Application Communities. In *Proceedings of the* 13^{th} *Symposium on Network and Distributed System Security (NDSS 2006)*, pages 95–106, February 2006.

[LSK06b] Michael E. Locasto, Angelos Stavrou, and Angelos D. Keromytis. Dark Application Communities. In *Proceedings of the* 15^{th} *New Security Paradigms Workshop (NSPW)*, pages 11–18, September 2006.

[LWKS05] Michael E. Locasto, Ke Wang, Angelos D. Keromytis, and Salvatore J. Stolfo. FLIPS: Hybrid Adaptive Intrusion Prevention. In *Proceedings of the* 8^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 82–101, September 2005.

[MRVK07] Darren Mutz, William Robertson, Giovanni Vigna, and Richard Kemmerer. Exploiting Execution Context for the Detection of Anomalous System Calls. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2007.

[MS05] David J. Malan and Michael D. Smith. Host-Based Detection of Worms through Peer-to-Peer Cooperation. In *Proceedings of the* 3^{rd} *ACM Workshop on Rapid Malcode (WORM)*, November 2005.

[MSVS03] D. Moore, C. Shannon, G. Voelker, and S. Savage. Internet Quarantine: Requirements for Containing Self-Propagating Code. In *Proceedings of the IEEE Infocom Conference*, April 2003.

[MVVK06] Darren Mutz, Fredrik Valeur, Giovanni Vigna, and Christopher Kruegel. Anomalous System Call Detection. *ACM Transactions on Information and System Security*, 9(1):61–93, February 2006.

[NBS06] James Newsome, David Brumley, and Dawn Song. Vulnerability–Specific Execution Filtering for Exploit Prevention on Commodity Software. In *Proceedings of the* 13^{th} *Symposium on Network and Distributed System Security (NDSS 2006)*, February 2006.

[NKS05] James Newsome, B. Karp, and Dawn Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2005.

[NS03] Nicholas Nethercote and Julian Seward. Valgrind: A Program Supervision Framework. In *Electronic Notes in Theoretical Computer Science*, volume 89, 2003.

[NS05] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the* 12^{th} *Symposium on Network and Distributed System Security (NDSS)*, February 2005.

[nul] `http://www.securityfocus.com/bid/5774`.

[NZ06] S. Neuhaus and A. Zeller. Isolating Intrusions by Automatic Experiments. In *Proceedings of the ISOC Symposium on Network and Distributed System Security (SNDSS)*, pages 71–80, February 2006.

[NZZ07] Stephan Neuhaus, Thomas Zimmermann, and Andreas Zeller. Predicting Vulnerable Software Components. In *Proceedings of the* 14^{th} *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[OBM06] Xinming Ou, Wayne F. Boyer, and Miles A. McQueen. A Scalable Approach to Attack Graph Generation. In *Proceedings of the* 13^{th} *ACM Conference on Computer and Communications Security (CCS)*, October 2006.

[OL02] Jeffrey Oplinger and Monica S. Lam. Enhancing Software Reliability with Speculative Threads. In *Proceedings of the* 10^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, October 2002.

[OLHL02] A. Orso, D. Liang, M. J. Harrold, and R. Lipton. Gamma System: Continuous Evolution of Software After Deployment. In *Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA)*, July 2002.

[OS04] A. J. O'Donnell and H. Sethu. On Achieving Software Diversity for Improved Network Security using Distributed Coloring Algorithms. In *Proceedings of the* 11^{th} *ACM Conference on Computer and Communications Security (CCS)*, pages 121–131, October 2004.

[OSSN02] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In

Proceedings of the 5^{th} *Symposium on Operating Systems Design and Implementation (OSDI 2002)*, pages 361–376, December 2002.

[Ove98] Richard E. Overill. How Re(Pro)active Should an IDS Be? In *Proceedings of the* 1^{st} *International Workshop on Recent Advances in Intrusion Detection (RAID)*, September 1998.

[PAM06] Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. Network-Level Polymorhpic Shellcode Detection Using Emulation. In *Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2006.

[PF95] H. Patil and C. N. Fischer. Efficient Turn-time Monitoring Using Shadow Processing. In *Proceedings of the* 2^{nd} *International Workshop on Automated and Algorithmic Debugging*, 1995.

[PFMA] Nick L. Petroni, Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot – a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the* 13^{th} *USENIX Security Symposium*, pages 179–194.

[Pie04] Tadeusz Pietraszek. Using Adaptive Alert Classification to Reduce False Positives in Intrusion Detection. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2004.

[PMM+07] Niels Provos, Dean McNamee, Panayiotis Mavrommatis, Ke Wang, and Nagendra Modadugu. The Ghost in the Browser: Analysis of Web-based Malware. In *HotBots 2007*, 2007.

[Pre99] Vassilis Prevelakis. A Secure Station for Network Monitoring and Control. In *Proceedings of the* 8^{th} *USENIX Security Symposium*, August 1999.

[Pro03] Niels Provos. Improving Host Security with System Call Policies. In *Proceedings of the* 12^{th} *USENIX Security Symposium*, pages 207–225, August 2003.

[QTSZ05] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating Bugs as Allergies – A Safe Method to Survive Software Failures. In

Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP), 2005.

[RCD+04a] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and Jr. W Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings* 6^{th} *Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[RCD+04b] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel Roy, and Tudor Leu. A Dynamic Technique for Eliminating Buffer Overflow Vulnerabilities (and Other Memory Errors). In *Proceedings* 20^{th} *Annual Computer Security Applications Conference (ACSAC) 2004*, December 2004.

[RJCM03] James C. Reynolds, James Just, Larry Clough, and Ryan Maglich. On-Line Intrusion Detection and Attack Prevention Using Diversity, Genrate-and-Test, and Generalization. In *Proceedings of the* 36^{th} *Hawaii International Conference on System Sciences (HICSS)*, 2003.

[RN02] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2002.

[RW01] A. Rudys and D. S. Wallach. Transactional Rollback for Language-Based Systems. In *ISOC Symposium on Network and Distributed Systems Security (SNDSS)*, February 2001.

[RW02] A. Rudys and D. S. Wallach. Termination in Language-based Systems. *ACM Transactions on Information and System Security*, 5(2), May 2002.

[SC05] A. Smirnov and T. Chiueh. DIRA: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks. In *Proceedings of the* 12^{th} *Symposium on Network and Distributed System Security (NDSS)*, February 2005.

[SCM04] Andersson Stig, Andrew Clark, and George Mohay. Network-based Buffer Overflow Detection by Exploit Code Analysis. In *AusCERT Conference*, May 2004.

[SEVS04] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated Worm Fingerprinting. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[SF00] A. Somayaji and S. Forrest. Automated Response Using System-Call Delays. In *Proceedings of the* 9^{th} *USENIX Security Symposium*, August 2000.

[SGK05] Stelios Sidiroglou, Giannis Giovanidis, and Angelos D. Keromytis. A Dynamic Mechanism for Recovering from Buffer Overflow Attacks. In *Proceedings of the* 8^{th} *Information Security Conference (ISC)*, pages 1–15, September 2005.

[SGP+02] John D. Strunk, Garth R. Goodson, Adam G. Pennington, Craig Soules, and Gregory Ganger. Intrusion Detection, Diagnosis, and Recovery with Self-Securing Storage. Technical Report CMU-CS-02-140, CMU Computer Science, May 2002.

[SHJ+02] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J.M. Wing. Automated Generation and Analysis of Attack Graphs. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2002.

[SK03] Stelios Sidiroglou and Angelos D. Keromytis. A Network Worm Vaccine Architecture. In *Proceedings of the IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Workshop on Enterprise Security*, pages 220–225, June 2003.

[SLBK05] Stelios Sidiroglou, Michael E. Locasto, Stephen W. Boyd, and Angelos D. Keromytis. Building a Reactive Immune System for Software Services. In *Proceedings of the USENIX Annual Technical Conference*, pages 149–161, April 2005.

[SLKN07] Stelios Sidiroglou, Oren Laadan, Angelos D. Keromytis, and Jason Nieh. Using Rescue Points to Navigate Software Recovery (Short Paper). In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 273–278, May 2007.

[SLS+07] Yingbo Song, Michael E. Locasto, Angelos Stavrou, Angelos D. Keromytis, and Salvatore J. Stolfo. On the Infeasibility of Modeling Polymorphic Shellcode. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 541–551, 2007.

[SLZD04] G. Edward Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the* 11^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)*, October 2004.

[Spi03] Diomidis Spinellis. Reliable identification of bounded-length viruses is NP-complete. *IEEE Transactions on Information Theory*, 49(1):280–284, January 2003.

[SPP+04] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and Dan Boneh. On the Effectiveness of Address-Space Randomization. In *Proceedings of the* 11^{th} *ACM Conference on Computer and Communications Security (CCS)*, pages 298–307, October 2004.

[SS98] Christopher Small and Margo Seltzer. MiSFIT: A Tool for Constructing Safe Extensible C++ Systems. *IEEE Concurrency*, 6(3):33–41, 1998.

[Sta04] Mark Stamp. Risks of Monoculture. *Communications of the ACM*, 47(3):120, March 2004.

[Sto04] S. Stolfo. Worm and Attack Early Warning: Piercing Stealthy Reconnaissance. *IEEE Privacy and Security*, pages 73–75, May/June 2004.

[sub02] The SUBTERFUGUE Project. `http://subterfugue.org/`, April 2002.

[TE04] Dean Turner and Stephen Entwisle. Symantec Internet Security Threat Report. `http://enterprisesecurity.symantec.com/content.cfm?articleid=1539`, September 2004.

[TG06] Carol Taylor and Carrie Gates. Challenging the Anomaly Detection Paradigm: A Provocative Discussion. In *Proceedings of the* 15^{th} *New Security Paradigms Workshop (NSPW)*, pages 21–29, September 2006.

[TK02] Thomas Toth and Christopher Kruegel. Accurate Buffer Overflow Detection via Abstract Payload Execution. In *Proceedings of the* 5^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 274–291, October 2002.

[TLH+07] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, Yuanyuan Zhou, James Newsome, David Brumley, and Dawn Song. Sweeper: A Lightweight End-to-End System for Defending Against Fast Worms. In *EuroSys*, 2007.

[WCS05] Ke Wang, Gabriela Cretu, and Salvatore J. Stolfo. Anomalous Payload-based Worm Detection and Signature Generation. In *Proceedings of the* 8^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 227–246, September 2005.

[WFP03] Nicholas Wang, Michael Fertig, and Sanjay J. Patel. Y-Branches: When You Come to a Fork in the Road, Take It. In *Proceedings of the* 12^{th} *International Conference on Parallel Architectures and Compilation Techniques*, September 2003.

[WGSZ04] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In *Proceedings of the ACM SIGCOMM*, August 2004.

[Whi03] J. A. Whittaker. No Clear Answers on Monoculture Issues. *IEEE Security & Privacy*, 1(6):18–19, November/December 2003.

[WK03] John Wilander and Mariam Kamkar. A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS)*, 2003.

[WLK06] Benny Wong, Michael E. Locasto, and Angelos D. Keromytis. PalProtect: A Collaborative Security Approach to Comment Spam. In *Proceedings of the IEEE Information Assurance Workshop (IAW)*, pages 170–175, June 2006.

[WLX+06] XiaoFeng Wang, Zhuowei Li, Jun Xu, Michael K. Reiter, Chongkyung Kil, and Jong Youl Choi. Packet Vaccine: Black-box Exploit Detection and Signature Generation. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 37–46, 2006.

[WPLZ06] Xinran Wang, Chi-Chun Pan, Peng Liu, and Sencun Zhu. SigFree: A Signature-free Buffer Overflow Attack Blocker. In *Proceedings of the* 15^{th} *USENIX Security Symposium*, pages 225–240, 2006.

[WPS06] Ke Wang, Janak J. Parekh, and Salvatore J. Stolfo. ANAGRAM: A Content Anomaly Detector Resistant To Mimicry Attack. In *Proceedings of the* 9^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 226–248, 2006.

[WS02] David Wagner and Paolo Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, November 2002.

[WS04] Ke Wang and Salvatore J. Stolfo. Anomalous Payload-based Network Intrusion Detection. In *Proceedings of the* 7^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 203–222, September 2004.

[XKI03] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent Runtime Randomization for Security. In *Proceedings of the* 22^{nd} *International Symposium on Reliable Distributed Systems (SRDS)*, 2003.

[XNK+05] Jun Xu, Peng Ning, Chongkyung Kil, Yan Zhai, and Chris Bookholt. Automatic Diagnosis and Response to Memory Corruption Vulnerabilities. In *Proceedings of the* 12^{th} *ACM Conference on Computer and Communications Security (CCS)*, November 2005.

[YGBJ05] Vinod Yegneswaran, Johnathon T. Giffin, Paul Barford, and Somesh Jha. An Architecture for Generating Semantics-Aware Signatures. In *Proceedings of the* 14^{th} *USENIX Security Symposium*, 2005.