# SYSTEM SUPPORT FOR IMPROVING SOFTWARE DEPENDABILITY DURING PRODUCTION RUNS

BY

FENG QIN

B.E., University of Science and Technology of China, 1998
M.E., Academia Sinica, 2001

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois

# Abstract

As hardware performance and dependability have dramatically improved in the past few decades, the software dependability issues are becoming increasingly important. Unfortunately, many studies show that software bugs, which inevitably slip through various bug detection methods and even the strictest testing before releasing, can greatly affect software dependability during production runs. To improve software dependability during production runs, this dissertation proposes to address software bugs at multiple levels by leveraging support from the underlying hardware, the OS kernel, and the middle-layer runtime.

The proposed multi-level defenses address software bugs and their effects at different stages of program execution. The first-level defense detects software bugs once they are triggered. The detection at the earliest stage can effectively prevent further propagation of errors that are caused by the software bugs. It would be perfect if we could detect all the software bugs at the first-level defense. However, some bugs may still slip through the first-level defense and may be exploited by security attacks. The second-level defense is to detect the exploitation of software bugs in order to control the system damage caused by the potentially exploited bugs. Due to the limitation of the tools or methods deployed in the first-level and second-level defenses, some bugs may still escape them. Additionally, without any further actions for the detected bugs or exploitations at the previous two levels of defenses, what the target system can do is to shut down itself to prevent potential damages, thus is unavailable to users. At this point, the third-level defense recovers the program from software bugs and their effects, thus providing non-stop services. In short, the multi-level defenses complement each other to effectively address software bugs during production runs.

More specifically, in each level of defense, this dissertation proposes a novel low-overhead

method to address software bugs during production runs by leveraging support from the hardware, OS, or the runtime. In the first-level defense, this dissertation proposes a low-overhead tool, called SafeMem, to detect memory leaks and memory corruption bugs, two major forms of software bugs that severely threaten system availability and security. It does not require any new hardware extensions. Instead, SafeMem makes a novel use of existing ECC memory technology and exploits intelligent dynamic memory usage behavior analysis to detect memory leak and corruption bugs. The experiments with seven real-world applications show that SafeMem detects all tested bugs with very low overhead (only 1.6%-14.4%), 2-3 orders of magnitudes smaller than Purify, a well-known bug detection tool.

In the second-level defense, this dissertation proposes a low-overhead, software-only information flow tracking system, called LIFT, to detect the exploitation of software bugs. Without requiring any hardware changes, LIFT minimizes runtime overhead by exploiting dynamic binary translation and optimizations for detecting various types of security attacks. More specifically, LIFT aggressively eliminates unnecessary dynamic information tracking, coalesces information checks, and efficiently switches between target programs and instrumented information flow tracking code. The experiments with two real-world server applications, one client application and eighteen attack benchmarks show that LIFT can effectively detect various types of security attacks. LIFT also incurs very low overhead, only 6.2% for server applications, and 3.6 times on average for seven SPEC INT2000 applications. The proposed dynamic optimizations effectively reduce the overhead by a factor of 5-12 times.

In the third-level defense, this dissertation proposes an innovative technique, called Rx, which can quickly recover programs from many types of software bugs, both deterministic and non-deterministic. The idea, inspired from allergy treatment in real life, is to roll back the program to a recent checkpoint once failure, triggering or exploitation of software bugs that are detected at the first two level of defenses, and then re-execute the program in a modified environment. This idea is based on the observation that many bugs are correlated with their execution environments, and therefore can be avoided by removing the "allergen" from the environment. Rx requires few to no

modification to applications and provides programmers with additional feedback for bug diagnosis. The experiments with four server applications that contain six bugs of various types show that Rx can survive all the six software failures and provide transparent fast recovery within 0.017-0.16 seconds, 21-51 times faster than the whole system program restart approach for all but one case (CVS).

*To my wife, Yuan,*

*and my parents, Shouren and Xiaoqiu*

# Acknowledgments

I owe biggest thanks to my adviser Yuanyuan Zhou. Every time I need her feedback or suggestion, she would always be there. She inspires and influences me not only on research, but also on many other aspects of my life. She plays a key role at two turning points of my life. I will never forget her words: “Do not look at the bar, look at the sky!”

Thanks also go to the other members of my dissertation committee, Professors William Sanders, Lui Sha, and Josep Torrellas, for their time and insightful feedback. I would like to thank Professor Darko Marinov and Indranil Gupta for insightful comments on the presentation of my work.

I thank all members in the Opera group. I enjoyed the inspiring research atmosphere. I would pay special tribute to Joseph Tucek, Zhenmin Li, Shan Lu, and Pin Zhou, whom I happily collaborated with in several projects. Zhifeng Chen, Qingbo Zhu, Weihang Jiang, Xiao Ma, Lin Tan, and Vivek Pandey bounced ideas with me and provided valuable comments on my papers.

I would like to thank researchers in Intel labs, Youfeng Wu, Cheng Wang, Ho-seop Kim, Chen Yang, and Xiaofeng Li for discussing many research ideas with me and providing me feedback on the LIFT project. I also want to extend thanks to Cezary Dubnicki in NEC research lab for his kindness in helping me on my intern project then.

I could not thank enough my dearest parents in China. I spent my precious eighteen years with them and shared many exciting moments with them. They taught me many things during those years. They used to and will always support me. I would like to thank my brother and sister-in-law, who take care of my parents in China and compensate a lot for my absence.

Last but not least, special thanks to my dearest wife Yuan, for her enormous support. She made my PhD process a pleasant journey!

# Table of Contents

**List of Tables** . . . . . **xi**

**List of Figures** . . . . . **xii**

**Chapter 1 Introduction** . . . . . **1**
1.1 Software Dependability During Production Runs . . . . . 1
1.1.1 Dependability is Important . . . . . 1
1.1.2 Software Bugs Affect Dependability During Production Runs . . . . . 2
1.1.3 Software Bugs Inevitably Exist in Production Runs . . . . . 3
1.1.4 Scope of Software Bugs in This Dissertation . . . . . 4
1.2 Addressing Software Bugs During Production Runs . . . . . 5
1.2.1 The Evolution Process of Triggered Software Bugs . . . . . 5
1.2.2 Desired Characteristics of Online Tools for Fighting Software Bugs . . . . 6
1.2.3 State of The Art . . . . . 7
1.3 Dissertation Contributions . . . . . 10
1.3.1 Multi-Level Defenses for Software Bugs . . . . . 10
1.4 Outline . . . . . 13

**Chapter 2 Background and Related Work** . . . . . **14**
2.1 Software Bug Detection . . . . . 14
2.2 Software Bug Exploitation Detection . . . . . 17
2.3 Surviving Software Failures . . . . . 18

**Chapter 3 First-Level Defense: Detecting Memory Bugs** . . . . . **21**
3.1 Overview . . . . . 21
3.2 Low-Overhead Monitoring of Memory Accesses . . . . . 23
3.2.1 ECC Background . . . . . 23
3.2.2 Using ECC to Monitor Memory Accesses . . . . . 24
3.3 Detecting Memory Leaks . . . . . 29
3.3.1 Characteristics and Classification of Continuous Memory Leaks . . . . . 30
3.3.2 Detection Process . . . . . 32
3.4 Detecting Memory Corruptions . . . . . 35
3.5 Methodology . . . . . 36
3.5.1 Platform . . . . . 36
3.6 Results . . . . . 38

3.6.1 Microbenchmark Results . . . 38
3.6.2 Overall Results . . . 38
3.6.3 Benefits of ECC Protection . . . 39
3.6.4 Effects of ECC-Protection in False Pruning for Memory Leaks . . . 40
3.7 Summary . . . 41

**Chapter 4 Second-Level Defense: Detecting Bug Exploits . . . 42**
4.1 Overview . . . 42
4.1.1 Motivation . . . 42
4.1.2 Highlight of LIFT . . . 44
4.2 LIFT Basic Design and Implementation . . . 46
4.2.1 Dynamic Binary Instrumentation Framework . . . 47
4.2.2 Tag Management . . . 47
4.2.3 Information Flow Tracking . . . 49
4.2.4 Exploit Detection . . . 49
4.2.5 An Example of Information Flow Tracking for LIFT-basic . . . 50
4.2.6 Protection of Tag Space and LIFT Code. . . . 50
4.3 LIFT Binary Optimizations . . . 51
4.3.1 Fast Path (FP) Optimization . . . 52
4.3.2 Merged Check (MC) Optimization . . . 55
4.3.3 Fast Switch (FS) Optimization . . . 56
4.4 Evaluation Methodology . . . 57
4.5 Experimental Results . . . 59
4.5.1 Security Attack Detection . . . 59
4.5.2 Performance Results . . . 61
4.6 Summary . . . 65

**Chapter 5 Third-Level Defense: Surviving Software Failures . . . 66**
5.1 Overview . . . 66
5.1.1 Motivation . . . 66
5.1.2 Highlights of Rx . . . 68
5.2 Main Idea of Rx . . . 71
5.3 Rx Design . . . 76
5.3.1 Sensors . . . 76
5.3.2 Checkpoint and Rollback . . . 77
5.3.3 Environment Wrappers . . . 80
5.3.4 Proxy . . . 83
5.3.5 Control Unit . . . 86
5.4 Design and Implementation Issues . . . 87
5.5 Evaluation Methodology . . . 89
5.6 Experimental Results . . . 91
5.6.1 Overall Results . . . 91
5.6.2 Recovery Performance . . . 94
5.6.3 Rx Time and Space Overhead . . . 95

5.6.4 Benefits of the Failure Table . . . . . 97
5.7 Summary . . . . . 98

**Chapter 6 Conclusions and Future Work . . . . . 99**

**References . . . . . 102**

**Vita . . . . . 113**

# List of Tables


3.1 Tested applications . . . . . 37
3.2 Time for the ECC system calls . . . . . 38
3.3 Time overhead (%) comparison between SafeMem and Purify . . . . . 39
3.4 Space overhead (%) comparison between ECC- and page-protection approaches . . 40
3.5 False memory leaks reported before and after using ECC-protection . . . . . 40

4.1 An example of information flow tracking . . . . . 43
4.2 Applications and security exploits . . . . . 58
4.3 Results of LIFT for attack benchmarks . . . . . 59
4.4 Throughput and Response Time of Apache . . . . . 60

5.1 Possible environmental changes and their potentially-avoided bugs . . . . . 73
5.2 Applications and bugs . . . . . 89
5.3 Overall results: comparison of Rx and two alternative approaches . . . . . 91
5.4 The average space overhead per checkpoint . . . . . 97

# List of Figures


1.1 Methods of reducing software bugs prior to releasing software . . . . . 3
1.2 The Evolution Process of Triggered Software Bugs . . . . . 5

3.1 Read/Write operations for ECC memory . . . . . 22
3.2 Implementation of WatchMemory . . . . . 26
3.3 Stability of maximal lifetime . . . . . 29

4.1 Information flow tracking for three different instructions in LIFT-basic . . . . . 51
4.2 Distribution of four groups of tag propagation in Apache . . . . . 53
4.3 An example of the FP and MC optimizations. . . . . 54
4.4 Overall Results . . . . . 62
4.5 The check version execution percentage for SPEC . . . . . 63
4.6 Memory Check Number Reduction for SPEC . . . . . 64

5.1 Rx: The main idea . . . . . 72
5.2 Rx architecture . . . . . 76
5.3 Proxy in normal and recovery mode . . . . . 83
5.4 Throughput and average response time of Squid . . . . . 94
5.5 Throughput and average response time with different bug arrival rates . . . . . 95
5.6 Rx overhead with different checkpoint intervals . . . . . 96
5.7 Rx recovery time for the first and subsequent bug occurrences . . . . . 97

# Chapter 1

# Introduction

As hardware performance and dependability have dramatically improved in the past few decades, the software dependability issues are becoming increasingly important. Unfortunately, many studies show that software bugs, which inevitably slip through various bug detection methods and even the strictest testing before releasing, can greatly affect software dependability during production runs. According to the National Institute of Standards and Technology, software bugs cost the U.S. economy an estimated $59.5 billion annually, or approximately 0.6% of the gross domestic product!

To improve software dependability during production runs, this dissertation proposes to address software bugs at multiple levels by leveraging support from the underlying hardware, the OS kernel, and the middle-layer runtime.

## 1.1 Software Dependability During Production Runs

### 1.1.1 Dependability is Important

High dependability [ALRL04] is demanded by many programs, including both low-end and high-end software. For low-end software such as our daily use calendars and email clients, low dependability, e.g., frequent program crashes, may frustrate and annoy users, thus lead to business losses. Even more important is dependability for high-end software such as mission-critical applications and many Internet servers since they affect millions of users at the same time and can lose millions of dollars in hours or even minutes. According to a report from Gartner Group [Sco98], the cost of

one hour downtime of financial applications exceeds six millions US dollars. With the tremendous development of Internet and computer systems, almost every kind of organizations is becoming dependent upon highly dependable systems.

### 1.1.2 Software Bugs Affect Dependability During Production Runs

Unfortunately software bugs can greatly affect system dependability, e.g. availability, reliability, and security. A number of studies [Gra86, Gra90, Sco99, OGP03, MS00] in various types of computer systems have shown that software bugs are one of the major causes of system failures. For the deployed Tandem systems, Gray's report in 1986 [Gra86] shown that software bugs caused 25% of system failures. Later in 1990, when more data was available, Gray reported that software bugs caused 55% of system failures [Gra90]. Similarly, Oppenheimer et.al. studied three large-scale Internet servers in 2003 and reported that software bugs account for 24% of the root causes of system failures [OGP03].

More specifically, software bugs can hurt system availability and reliability. During program execution, the software bugs, triggered in the exercised execution path, may cause system crashes, hang, or other unexpected system misbehavior. There are many instances of software bugs affecting system availability and reliability in the real world. For example, the recently-happened Tokyo stock exchange system crash [Her05] in November 2005 is because of a software bug. It lasted for around four hours and paralyzed the whole trading business in Tokyo then.

Additionally, software bugs such as buffer overflow severely threaten computer system security. These bugs allow malicious users to launch security attacks by executing arbitrary code, causing denial of service, or stealing confidential data from a vulnerable system. For example, the fastest-ever worm, Slammer worm in 2003, exploited the buffer overflow bug in Microsoft SQL servers. It brought down tens of thousands of machines within several minutes and cost hundreds of millions of dollars loss [MPS+03].

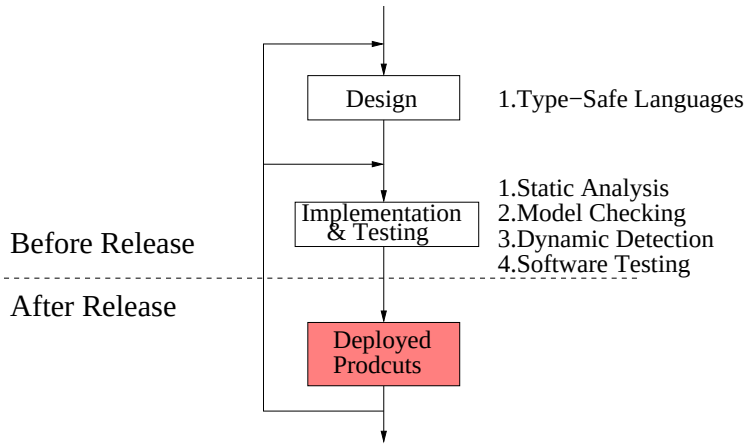



Figure 1.1: Methods of reducing software bugs prior to releasing software

### 1.1.3 Software Bugs Inevitably Exist in Production Runs

As Figure 1.1 shows, many approaches or techniques have been proposed to prevent or detect software bugs before software products are released. They can be classified into four categories. The approaches in the first category use type-safe languages such as Java or the Microsoft Common Language Runtime Environment [Mic] to prevent or alleviate certain types of software bugs such as memory leaks and buffer overflow. While these approaches can improve code quality, they are not applicable to performance-critical software such as server programs due to significant runtime overhead. In addition, type-safe languages usually do not allow fine-grained manipulation of data structures. As a result, most performance-critical software programs are still written in type-unsafe languages such as C or C++.

Static program analysis tools, such as LCLint [EGHT94], PREfix [BPS00], METAL [HCXE02], Clouseau [HL03], CSSV [DRS03], etc., belong to the second category. They scan the program source code at compile time and detect various types of software bugs such as memory leaks and NULL pointer dereference through program analysis techniques. While these tools do not impose runtime overheads, they may miss a lot of bugs and/or generate many false alarms because accurate runtime information is unavailable at compile time. Furthermore, some of these tools require annotations, which many programmers find too tedious.

The third category of tools leverages model checking techniques, such as SPIN [Hol97], Verisoft [God97],

CMC [MPC+02], etc, to detect subtle software bugs that may be triggered only under some intricate sequence of events. Model checking tools are effective to find those subtle software bugs by systematically exploring the possible system states with or without specification. However, the state explosion problem, a major limitation, results in undisclosed software bugs in the vast of unexplored states. Furthermore, the conventional model checking tools [Hol97] requires building an abstract specification, i.e., the model, which is a time-consuming and error-prone process.

The fourth category of approaches, software testing and dynamic detection tools such as Valgrind [NS03] and Purify [HJ92a], are common practice in software companies for detecting software bugs. While they can detect some software bugs, some still slip through even the strictest testing because of two reasons. First, they can only detect bugs on the exercised execution paths. However, it is nearly impossible to test all execution paths because the problem of generating a complete test suite that covers all execution paths is *theoretically uncomputable* [Rop94]. Second, many commonly used dynamic detection tools such as Purify [HJ92a] incur large runtime overheads, slowing down a program by up to 40 times [CHM+03a, ZLF+04]. This factor deteriorates the limited execution path coverage problem in software testing.

In summary, software bugs cannot be eliminated during software development processes and inevitably exist in released software products. Therefore, addressing software bugs during *production runs* is a crucial task for improving software dependability.

### 1.1.4 Scope of Software Bugs in This Dissertation

This dissertation does not aim for all types of software bugs during production runs. Before talking about the scope of software bugs addressed in this dissertation, we need a way to classify software bugs. So far there is no standard way to classify software bugs although several different ways [Bei90, LLC06] were proposed. As prior work [LLQ+05], this dissertation classifies software bugs into three categories: 1) memory-related bugs, which are caused by improper memory manipulation, including buffer overflow, memory leak, dangling pointer, etc. 2) concurrency bugs, which are related to incorrect synchronization among multiple threads or processes, including data

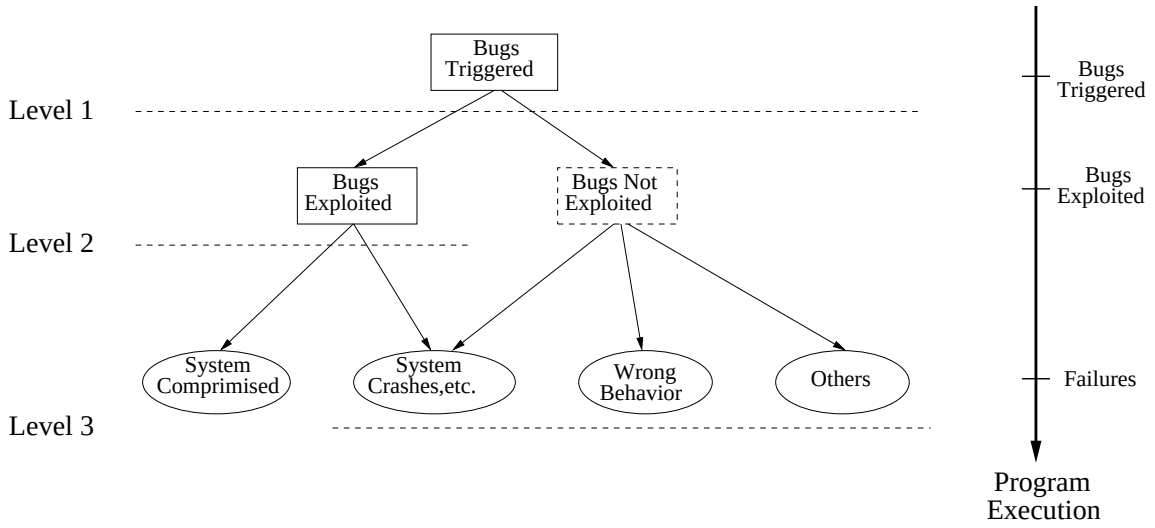



Figure 1.2: The Evolution Process of Triggered Software Bugs

race, deadlock, etc. 3) semantic bugs, which are incorrect implementation of the original design and may cause wrong functionality, for example, incorrect handling of corner cases.

This dissertation focuses on handling memory-related bugs although part of the work also deal with some concurrency bugs. The reason for memory-related bugs is that they are commonly happened ones and usually cause severe security problems. According to the US-CERT Vulnerability Notes Database [US-06], 39% of all reported vulnerabilities since 1991 were caused by memory leak or memory corruption, and 55% of the most severe vulnerabilities were related to them. In the year of 2003, these two types of memory-related bugs contributed to 68% of the CERT/CC [CER] advisories.

## 1.2 Addressing Software Bugs During Production Runs

### 1.2.1 The Evolution Process of Triggered Software Bugs

To address software bugs during production runs, we first need to understand the evolution process of software bugs as shown in Figure 1.2. At first, some software bug (or fault) is triggered during program execution. Here, the triggering of a software bug means executing the statement that

contains the software bug. After the triggering of a software bug, the program state becomes erroneous. In some fields such as fault tolerance, the triggering of a software bug indicates the transition from faults to errors. For example, a triggering of buffer overflow bug means that some memory area out of boundary is being accessed.

The triggered software bugs may or may not be exploited by malicious users for launching security attacks at some point later during program execution. Generally, the bug exploitation is a method to launch security attacks as long as the bug triggering is a necessary condition for the launched attacks. Here, we define the bug exploitation as the execution point when the program control flow is being switched to some unexpected program location that indicates the launch of security attacks. For example, stack smashing [One96] attacks usually exploit buffer overflow bugs to corrupt the return address with carefully-crafted program location. Once the function returns, the program will switch to some hijacked malicious code instead of the original parent function. At this point, we call the bug exploitation happening.

Along with program further execution, there are several possible results of the triggered software bugs, being either exploited or not exploited. If the software bug is exploited, the malicious users may do some harmful things such as compromising the target system, launching denial-of-service attacks, leaking sensitive information, building backdoors/Trojan [Tro06] for future attacks, etc. If the triggered software bug is not exploited by security attacks, the bug may manifest itself in various ways, such as, crashing/hanging the systems and delivering incorrect results to users, etc., or keep itself silent until the system normally terminates.

### 1.2.2 Desired Characteristics of Online Tools for Fighting Software Bugs

Addressing software bugs is a very challenging task. Different from in-house testing, the online tools should have some desired characteristics as follows.

- *Low overhead.* End-users usually are reluctant to deploy tools that may cause high runtime overhead to their production systems. For example, Purify [HJ92a], a very popular bug

detection tool, is only used for in-house testing instead of production runs because it can cause up to 40 times slowdown. This low overhead characteristic is especially important for server programs since bad server performance can affect millions of users at the same time.

- *No assumption on source code availability.* The tools or methods for production runs should be able to deal with binary code without relying on the help from source code. Usually, end-users only have binary distribution of the commercial software. Often times tools also need to deal with third-party libraries that end-users do not have source code. It makes the task more difficult without source code information as source code may provide useful information such as data types that are absent in binary code.

- *Low cost.* Our tools or methods should not rely on hardware extension since non-trivial hardware extension is expensive and takes time to implement. Some hardware approaches [ZQL+04, PT03] can improve software dependability while achieving good performance, while they are not available to existing systems due to non-trivial hardware extension.

### 1.2.3 State of The Art

A number of approaches have been proposed to improve software dependability during production runs by online detection of software bugs, online detection of security attacks that exploit software bugs, or recovery from software failures. The following part of this section summarizes the state-of-the-art such approaches and their limitations. More details are discussed in Section 2.

**Online detection of software bugs.** To overcome the large runtime overhead problem incurred by software-based dynamic bug detection tools such as Purify [HJ92a] and Valgrind [NS03], prior studies proposed several methods with the support from either static analysis technique [NMW02, CHM+03a, GMJ+02] or hardware extension [PT03, ZQL+04, ZLF+04]. CCured [NMW02] and Cyclone [GMJ+02] use static analysis to enforce a strong type system whenever possible and insert necessary runtime checks where static analysis cannot guarantee the type system. However, these

tools require pointer-object associations in order to check safety of pointer references. In addition, they require non-trivial changes to applications' source code to conform to their C standards. Other methods such as iWatcher [ZQL+04] and AccMon [ZLF+04] propose hardware extension for detecting software bugs. Although they usually incur very low runtime overhead, requiring non-trivial hardware extension makes them not applicable for existing systems.

**Online detection of security attacks.** Many tools or techniques [CPM+98, CBBKH01, CBJW03, sta06, BST00, XKPI02] were proposed to detect specific types of security attacks such as stack smashing [One96] and format string [tf800] at runtime. However, they are limited to those specific types of security attacks. Program randomization [XKI03] transforms system-compromising attacks to probable system crashes by diversifying program software layout at load time. Program shepherding [KBA02] leverages dynamic binary translation mechanisms to enforce some general security policies at runtime. Statistical intrusion detection methods [Far03] capture various invariants such as system call sequence invariants and mark violation of those invariants as security attacks. Although the program randomization, program shepherding, and statistical methods are not limited to several specific types of security attacks, they provide little information regarding the attacks, e.g., what are the attack input signatures, what are the attack steps, etc. This information is essential for building security rules in the intrusion detection and prevention systems such as Snort [Sno06].

Several recent work [CCC+05, NS05, SLZD04] demonstrated that information flow tracking is a promising and effective technique to effectively detect a wide range of security attacks. Generally this technique tags (labels) the input data from unsafe channels such as network connections as "unsafe" data, propagates the data tags through the computation (any data derived from unsafe data are also tagged as unsafe), and detects unexpected usages of the unsafe data that switch the program control to the unsafe data as exemplified in the stack smashing attack. However, either high runtime overhead or dependence on hardware extension makes existing information flow tracking systems unsuitable for production runs of existing systems.

**Online recovery from failures.** To increase software availability during production runs, many approaches are proposed to recover programs from failures caused by software bugs. Micro-rebooting [CCF+02, CKF+04] proposed to reboot the failed components instead of the whole system to minimize the system unavailable time. However, they cannot deal with deterministic bugs such as memory bugs since the deterministic bug will occur again after restart given the same input. Some application-specific methods such as multi-process model used by Apache [Apa06] and n-version programming [AC77, Avi85, RCL01] require non-trivial restructuring to existing programs. Additionally, n-version programming is prohibitively expensive so that it is unaffordable for most of the applications. Ammann and Knight's diversity-data system [AK98] may be useful for non-deterministic bugs, it typically cannot survive deterministic bugs such as buffer overflow and double free. For example, the related points of a very long input string are likely long so that they will still cause buffer overflow. Additionally, the diversity-data system method is not suitable for many applications that do not have accurate specifications for generating logically-equivalent input data points. Recently, Rinard et al. proposed failure-oblivious computing to recover programs from buffer overflow bugs and Sidiroglou et al. proposed reactive immune system [SLBK05] to confine errors within a certain program region such as a function. In addition, Demsky et al. [DR03, DR05] proposed to online repair fatal errors of important data structures via static specification or AI methods. While these approaches are fascinating and may work for certain types of applications, they are unsafe to use for correctness-critical applications (e.g. online banking systems) because they "speculate" on programmers' intentions, which can lead to program misbehavior.

In summary, prior work made progress on improving software dependability by addressing software bugs during production runs. However, they still suffer several limitations: requiring non-trivial modifications on source code or existing hardware, too specific for certain attacks or no enough information about attack signatures, and unable to address software bugs or unsafe speculation on program execution.

## 1.3 Dissertation Contributions

During program execution, once a software bug (fault) is triggered, we usually have a short period of time to handle it before it may cause system misbehaviors such as crashes or producing wrong results. Prior study [GKIY03] on Linux kernel's behavior under errors shows that nearly 40% of crash latencies are within 10 cycles and around 80% of crash latencies are within 100,000 cycles.

To improve software dependability during production runs, this dissertation proposes to address software bugs at multiple levels by leveraging support from the underlying hardware, the OS kernel, and the middle-layer runtime.

### 1.3.1 Multi-Level Defenses for Software Bugs

As shown in Figure 1.2, to maximize the chance of improving software dependability during production runs, this dissertation proposes to address software bugs at multiple levels by leveraging support from the underlying hardware, the OS kernel, and the middle-layer runtime.

**The First-Level Defense.** Once a software bug is triggered in the exercised execution path, it would be the most effective if we can catch it on the spot since we can prevent the system state error, which caused by the triggered software bug, from further propagation and affecting the whole system state. For example, we can prevent the corrupted memory data from further polluting other data once we catch the buffer overflow bug on accessing the out-of-bound memory. Additionally, detecting software bugs at this early stage could provide more detailed information about the software bug itself, usually the root cause, for program recovery and further diagnosis. For example, the more detailed information about a buffer overflow could be which memory buffer is overflowed and which instructions cause this overflow.

At this level, this dissertation proposes a low-overhead tool, called SafeMem, to detect memory leaks and memory corruption bugs, two major forms of software bugs that severely threaten system availability and security. Instead of relying on any new hardware extension, SafeMem makes

a novel use of existing ECC memory technology to provide low-overhead, fine-grained memory monitoring functionality to user-level applications. Combined with the proposed intelligent dynamic memory usage behavior analysis, SafeMem can detect memory leaks and memory corruption bugs with very low overhead and few to no false positives. Furthermore, SafeMem requires no modification to applications. Our evaluation of SafeMem with seven real-world applications that contain memory leak and memory corruption bugs show that SafeMem detects all the tested bugs with very low overhead (only 1.6%-14.4%), 2-3 orders of magnitudes smaller than Purify, a well-known bug detection tool. This indicates that SafeMem can be deployed for the first-level defense for production runs.

**The Second-Level Defense.** Unfortunately, some software bugs may still slip through the detection deployed in the first-level defense. For example, SafeMem cannot deal with double free bugs, buffer overflow bugs of local variables in the stack, etc. These software bugs could potentially be exploited by malicious users for launching security attacks. Therefore, it is essential to deploy the second-level defense to catch exploitation of software bugs since at least it can help to control the system damage that can potentially be caused by security attacks. Additionally, this level defense may provide useful information regarding to the malicious input such as the bug-exploiting input signatures, which can be used for filtering out future messages that match with attack signatures.

At this level, this dissertation proposes a low-overhead, software-only information flow tracking system, called LIFT, to detect the exploitation of software bugs. Without requiring any hardware changes, LIFT minimizes runtime overhead by exploiting dynamic binary translation and optimizations for detecting a wide range of security attacks. More specifically, LIFT aggressively eliminates unnecessary dynamic information flow tracking, coalesces information checks, and efficiently switches between target programs and instrumented information flow tracking code.

I implemented LIFT on top of a dynamic binary translation framework in Windows. The real-system experiments with two real-world server applications, one client application and eighteen attack benchmarks show that LIFT can effectively detect various types of security attacks. LIFT

also incurs very low overhead, only 6.2% for server applications, and 3.6 times on average for seven SPEC INT2000 applications. The proposed dynamic optimizations effectively reduce the overhead by a factor of 5-12 times.

**The Third-Level Defense.** Once software bugs or the exploitation are detected in the first-level or second-level defense, the system could be forcefully terminated to prevent system damage. Even worse, some software bugs may slip through the first two level defenses and cause system failures such as crashes. As mentioned in Section 1.1, low availability could mean a big loss of productivity and business. Therefore, at this point, it is important to deploy the third-level defense to recover programs from software bugs and their bad effects. As a result, the third-level defense helps increase service availability and provide non-stop services to users.

In this level, this dissertation proposes an innovative safe technique, called Rx, which can quickly recover programs from many types of software bugs, both deterministic and non-deterministic. The idea, inspired from allergy treatment in real life, is to rollback the program to a recent checkpoint once failure, triggering or exploitation of software bugs that are detected at the first two level defenses, and then re-execute the program in a modified environment. This idea is based on the observation that many bugs are correlated with their execution environment, and therefore can be avoided by removing the "allergen" from the environment. Rx requires few to no modification to applications and provides programmers with additional feedback for bug diagnosis.

We conducted experiments with four server applications that contain six bugs of various types, including heap buffer overflow, stack buffer overflow, double free, uninitialized read, dangling pointer, and data race. The results show that Rx can survive all the six software failures and provide transparent fast recovery within 0.017-0.16 seconds, 21-51 times faster than the whole system program restart approach for all but one case (CVS).

In summary, with support from the underlying hardware, the OS kernel, and the middle-layer runtime, this dissertation proposed multi-level defenses can effectively improve software dependability during production runs by addressing software bugs.

## 1.4 Outline

The remainder of the dissertation is organized as follows. Chapter 2 reviews prior work on software bug detection, software bug exploitation detection, and surviving software failures. Chapter 3 presents SafeMem in the first level defense, which uses ECC memory to detect memory leak and memory corruption bugs. Chapter 4 focuses on LIFT in the second level defense, a low overhead dynamic information tracking system to detect exploitation of software bugs. Chapter 5 discusses Rx in the third level defense, using system support to survive software failures during production runs. Chapter 6 summarizes this dissertation and discusses future research direction.

The materials in some chapters have been or will be published as conference papers. The materials in Chapter 3 have been presented in [QLZ05]. The materials in Chapter 4 will appear in [QWL+06]. The materials in Chapter 5 have been presented in [QTSZ05].

# Chapter 2

# Background and Related Work

This chapter reviews previous work on software bug detection, software bug exploitation detection, and surviving software failures.

## 2.1 Software Bug Detection

Researchers proposed many software bug detection tools or techniques, which can be classified into three categories: static program analysis, model checking, and dynamic detection. While tools in the first two categories can be used during software development, dynamic detection tools may be used before or after software deployment.

**Static program analysis.** Generally tools in this category [EGHT94, FTA02, DLS02, HCXE02, BPS00, DRS03, HL03] scan the program source code and use various static analysis techniques to detect potential software bugs. Evans's LCLint [EGHT94] detects the inconsistency between the source code and properties inferred from user-provided annotations. Foster et al. proposed CQual [FTA02], a general framework for checking program invariants specified by customized type qualifiers. Similarly, METAL [HCXE02] checks the source code and detect violations of programming rules, either provided by programmers [ECCH00] or automatically inferred from the source code itself [ECH+01]. Some tools are designed to detect certain types of bugs such as memory leaks and memory corruption. Clouseau [HL03] detects memory leaks using an object ownership and inference model, while CSSV [DRS03], proposed by Sagiv et al., detects unsafe string operations in C programs with the aid of procedure summaries.

While static tools do not impose runtime overheads, they may miss a lot of bugs and/or generate many false positives because accurate runtime information is unavailable at compile time. Recently proposed SAFECode [DKA06] by Dhurjati et al. can enforce alias analysis for weakly typed language, thus guarantee the soundness of some bug detection methods. However, SAFECode cannot help reduce many false positives generated by many static methods. Furthermore, some of these tools require annotations or summarizes, which impose extra burden on programmers. Besides common limitations, different static tools may have their own limitations. For example, Clouseau cannot handle type casting, pointer arithmetic, arrays of pointers, address of a pointer member field in a class or structure, concurrent execution and exception handling [HL03].

**Model checking.** Typically, model checking tools check system properties at the protocol level [McM93, Hol97, DDHY92] or the implementation level [God97, VHBP00, BMMR01, MPC+02] by exhaustively searching the system state space. Traditional model checking tools such as SMV [McM93] and SPIN [Hol97] focuses on verifying hardware and software protocols. Although they can detect non-trivial bugs, the requirement of building a model for the system in another language is the major drawback due to the required significant amount of manual effort that can easily lead errors.

Some recent software model checking tools such as Verisoft [God97] and CMC [MPC+02] systematically execute and check the systems in the implementation level. They have been used to check systems for concurrency bugs, e.g., deadlock, and assertion failures. Yang et al. applied the model checking method to widely-used, heavily-tested file systems and found serious software bugs [YTEM04]. Without requiring an abstract model for checking a target system is a big advantage. However, the state explosion problem is still a major obstacle for them, especially for large and complex systems, due to enormous state space need to be explored and limited computation resources.

**Dynamic detection.** Dynamic tools, either purely based on software [HJ92b, NS03, CHM+03b, ABS94, JK97, SBN+97] or relying on hardware extension [PT03, ZQL+04, ZLF+04], detect

software bugs at runtime. The state-of-the-art tools Purify [HJ92b] and Valgrind [NS03] detect memory-related bugs such as memory leaks and memory corruption by intercepting every memory access and monitoring every dynamically allocated memory objects through binary instrumentation. Jones and Kelly's tool [JK97], PointGuard [CBJW03], SafeC [ABS94] and CRED [RL04] can detect buffer overflows by dynamically checking each pointer dereference. While these tools do not suffer from the same limitation as static tools, most software-based detection tools incur high runtime overhead, up to 40 times [CHM+03a, ZLF+04] due to interception of every memory/pointer access.

Some hybrid schemes combine static and dynamic technique to alleviate the high overhead problem to some extent. CCured [NMW02, CHM+03a] is such a hybrid bug detection tool. It first attempts to enforce a strong type system in C programs via static analysis. Portions of the program that cannot be guaranteed by the CCured type system are instrumented with run-time checks to monitor the safety of executions. Cyclone [GMJ+02] is very similar. It changes the pointer representation to detect pointer dereference error. However, these tools require pointer-object associations in order to check safety of pointer references. They fail when such associations are not available (due to fine-grained pointer manipulation through various type-casting) or when the bug does not violate pointer-type/object association (such as a wrong pointer assignment bug caused by copy-paste). In addition, they require non-trivial changes to applications' source code to conform to their C standards.

Another direction to reduce runtime overhead incurred by software-based dynamic bug detection tools is to rely on hardware support. Some tools, such as ReEnact [PT03], iWatcher [ZQL+04], AccMon [ZLF+04], etc., propose hardware extension to support bug detection. Even though they usually incur very low runtime overhead, requiring non-trivial hardware extension to the existing microprocessor makes them not applicable for existing systems.

## 2.2 Software Bug Exploitation Detection

Many tools or techniques [CPM+98, CBBKH01, CBJW03, sta06, BST00, XKPI02] were proposed to detect specific types of security attacks such as stack smashing [One96] and format string [tf800] at runtime. Cowan et al. proposed StackGuard [CPM+98] for detecting stack smashing attacks by storing a special guard value in the added "canary" of the stack for each function and checking the guard value right before exiting the function. In addition to pure software approaches, Xu et al. proposed architecture-supported schemes [XKPI02] for defending stack smashing attacks. LibSafe [BST00] enhances some library functions that are known to be vulnerable with extra checks against buffer overflow, the direct cause of stack smashing attacks.

Some techniques [XKI03, Far03, KBA02] detect general types of security attacks through various mechanisms. Program randomization [XKI03] transforms system-compromising attacks to probable system crashes by shuffling program regions such as the stack region and the heap region in the memory during program load time. Thereby, once under attack, the program control flow jumps to some bizarre location and the program very likely crashes instead of being compromised. Statistical intrusion detection methods [Far03] capture various invariants such as system call sequence invariants and mark violation of those invariants as security attacks. Programming shepherding [KBA02] leverages dynamic binary translation mechanisms to enforce the security policies at runtime. All these techniques are not limited by some specific types of security attacks or software vulnerabilities. However, they provide little useful information regarding the attacks, for example, what are the attack input signatures, what are the attack steps, etc. This information is very useful for network-based applications to filter out future messages that match with attack signatures.

Recently, information flow tracking [Den76, DD77, HR98, Mye99, ML00, CCC+05, NS05, SLZD04, XBS06] has been demonstrated as a promising technique for detecting many system-compromising attacks, even for unknown types of exploits and software vulnerabilities. Generally, this technique tags the input data from unsafe channels such as network connections as "unsafe"

data, propagates the data tags through the computation, and detects unexpected usages of the unsafe data. Along this line, Vigilante [CCC+05] and TaintCheck [NS05] use dynamic binary instrumentation methods for the information flow tracking and incur large runtime overhead, up to 40x slowdown [NS05]. The hardware approach [SLZD04] proposed by Suh et al. can avoid the high overhead incurred by the pure software approaches. Xu et al. proposed a source code level instrumentation [XBS06] for the implementation that can alleviate high overhead problem to some extent. However, it cannot track information flow in third-party library code and thereby will miss security exploits involving these libraries.

## 2.3 Surviving Software Failures

Much research has been conducted on surviving software failures caused by deterministic bugs and non-deterministic bugs in the past decades.

**Surviving failures caused by non-deterministic bugs** Whole program restart [Gra86, SC91] is usually the first attempt to handle software failures with the hope that they are caused by non-deterministic bugs since non-deterministic bugs may disappear during re-execution. However, it may cause a long period of service unavailability [BBG+89, VDB+98] for server programs that buffer significant amount of state in main memory (e.g., data buffer caches). Software rejuvenation [HKKF95, GPTT97, BS98] is an interesting approach to reduce the period of unexpected service outage by rejuvenating/restarting the software to a fresh state after a certain period and before it fails. Recently Candea et al. proposed Micro-rebooting [CCF+02, CKF+04] to address this problem to some extent by only rebooting the failed components.

General checkpointing and recovery mechanisms [BBG83, EAWJ02, RLT78] have been proposed for surviving failures for a while. Typically, they checkpoint the program state, roll back the program upon failures, and then re-execute the program. The checkpoints may be done to disk [CPL97, JZ88, LNP90, WHV+95], non-volatile or persistent memory [LC98], or even re-

mote memory [ACZ00, ZCL99, PLP98]. These checkpoints can be provided with relatively low overhead. If there are messages and operations in flight, logging is also needed [Bir96, LC97, LCC00, JZ90]. To deal with resource exhaustion or operating system crashes, monitoring, logging and recovery can be done remotely via support by special network interface cards [BNG+04]. Wang et al. proposed progressive retry [WHF93], an interesting improvement over traditional checkpointing approaches. It increases the chance of surviving software failures that are related to message orders since reordering messages may increase non-determinism during re-execution. Ammann and Knight proposed a diversity-data system [AK98] to tolerate failures by increasing non-determinism via a set of related input data points that are logically equivalent. While manipulating input data may be useful for non-determinitistic bugs, it typically cannot survive deterministic bugs such as buffer overflow and double free. For example, the related points or reordering of a very long input string are likely long so that they will still cause buffer overflow.

Some application-specific recovery mechanisms such as the multi-process model (MPM) can be used to handle software failures caused by non-deterministic bugs. For example, the old version of the Apache HTTP server [Apa06] spawns a new process for each client connection and therefore can simply kill a failed process and start a new one to handle a failed request. It is simple, but requires programs to be failure-aware.

**Surviving failures caused by deterministic bugs** Various fault tolerance mechanisms can be used to survive software failures caused by deterministic bugs, a major cause of software failures [CC00]. The recovery blocks approach [HLMSR74, Ran75] extends a conventional block with a means of error detection and additional alternates, i.e., different implementation versions of the same block. The alternate will be executed upon a failure detected at the end of a block execution. Similarly, n-version programming [AC77, Avi85, RCL01] relies on different implementation versions of the same software unit. Unlike recovery blocks, it executes different versions concurrently and the result is a consensus result from all the versions. Both mechanisms can address software failures caused by deterministic bugs assuming that different implementation versions fail

independently. However, they are too expensive to be deployed in the normal software development process.

Two recently proposed, non-conventional approaches are failure-oblivious computing [RCD+04] and the reactive immune system [SLBK05]. Failure-oblivious computing proposes to deal with buffer overflows by providing artificial values for out-of-bound reads, while the reactive immune system returns a speculative error code for functions that suffer software failures. While these approaches are fascinating and may work for certain types of applications, they are unsafe to use for correctness-critical applications (e.g. on-line banking systems) because they "speculate" on programmers' intentions, which can lead to program misbehavior.

# Chapter 3

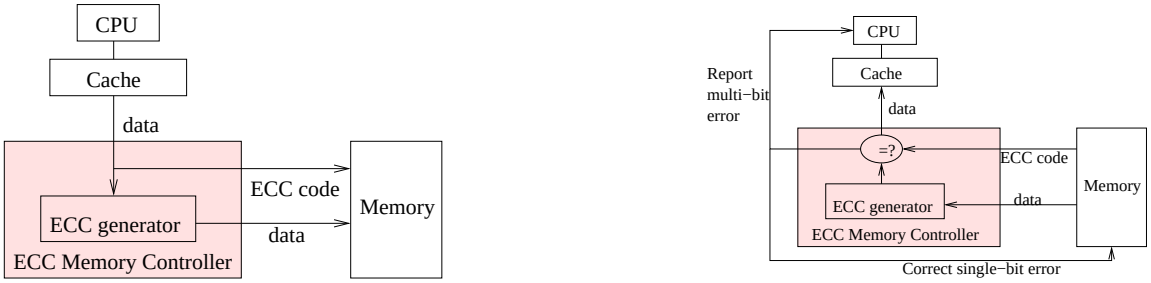
# First-Level Defense: Detecting Memory Bugs

## 3.1 Overview

The first-level defense is to detect software bugs once they are triggered during production runs. This chapter proposes a low-overhead tool, called SafeMem, to on-the-fly detect memory leaks and memory corruption bugs, two major forms of software bugs that severely threaten software reliability. According to the US-CERT Vulnerability Notes Database [US-06], 39% of all reported vulnerabilities since 1991 were caused by memory leaks or memory corruption, and 55% of the most severe vulnerabilities are related to them.

Memory leaks, caused when some allocated memory is never accessed again, can cumulatively degrade overall system performance by increasing memory paging. Even worse, they may cause programs to exhaust system resources, eventually leading to program crashes [HJ92b]. For this reason, malicious users often exploit memory leaks to launch denial-of-service attacks. Memory corruption, on the other hand, damages memory content through buffer overflow, incorrect pointer arithmetic, or other types of program errors. Similar to memory leaks, memory corruption bugs, especially buffer overflows, are commonly exploited by Internet attacks to attach malicious code through carefully-crafted input data.

As reviewed in Chapter 2, dynamic detection is commonly used by programmers to detect software bugs, including memory leaks and memory corruption. Dynamic detection can be performed either in software or with hardware support. Purify [HJ92b] is a state-of-the-art software-only dynamic tool for detecting memory leaks and memory corruption. However, Purify and most other software dynamic tools have a major limitation: incurring high run-time overhead. Sometimes




(a) *Write to ECC memory* (b) *Read from ECC memory*

Figure 3.1: Read/Write operations for ECC memory

these tools can slow down a program by up to 40 times [CHM+03a, ZLF+04]. Therefore, they cannot be used during production runs. iWatcher [ZQL+04] is a recently proposed architectural extension to reduce overheads for dynamic monitoring, but it requires hardware extensions and therefore cannot be used in existing systems.

This chapter proposes a low-overhead dynamic tool called SafeMem to detect memory leak and memory corruption *on-the-fly* during *production runs*. To reduce runtime overheads without relying on new hardware extension, it makes a novel use of existing Error-Correcting Code (ECC) memory technology and provides low overhead memory access monitoring. In addition, SafeMem exploits intelligent dynamic memory usage behavior analysis to detect memory leaks and use ECC-protection to prune false positives. To detect memory corruption bugs, SafeMem use ECC-protection to monitor illegal accesses, both to freed memory buffers and to the two ends of allocated memory buffers.

The rest of the chapter is organized as follows. Section 3.2 introduces the ECC memory and our novel use of this technology. Sections 3.3 and 3.4 present the methods to detect memory leaks and memory corruption, respectively, followed by the evaluation methodology in Section 3.5. Experiment results are presented in Section 3.6. Section 3.7 summarizes SafeMem.

## 3.2 Low-Overhead Monitoring of Memory Accesses

### 3.2.1 ECC Background

Error-Correcting Code (ECC) memory is commonly used in modern systems, especially server machines, to provide error detection and correction in case of hardware memory errors. It is an extension of simple parity memory, which can detect only single-bit errors. In contrast, ECC not only detects single-bit and multi-bit errors, but it also corrects single-bit errors on the fly, transparently. Unlike parity memory, which uses a single bit to provide protection to eight bits, ECC uses larger groupings: 7 bits to protect 32 bits, or 8 bits to protect 64 bits [Int]. For convenience, we call such a block of 32 bits or 64 bits an *ECC-group*. ECC requires special chipset support. When supported and enabled, ECC can function using ordinary parity memory modules; this is the standard way that most motherboards with ECC support operate. The chipset "groups" together the parity bits of memory modules into the 7 or 8-bit block needed for ECC.

Most ECC memory controllers support four modes: Disabled, Check-Only, Correct-Error and Correct-and-Scrub. In the Disabled mode, the memory controller disables all the ECC functionalities. In the Check-Only mode, the memory controller detects and reports single-bit and multi-bit errors, but it does not correct them. With the Correct-Error mode enabled, the memory controller not only detects single-bit and multi-bit errors, but it also corrects single-bit errors. This mode improves data integrity by seamlessly correcting single-bit errors. With the Correct-and-Scrub mode enabled, the memory controller not only detects and corrects errors, but it also scrubs memory periodically to check and correct hardware errors. This mode provides the highest data integrity.

ECC memory works as shown in Figure 3.1. At a write to memory, the memory controller encodes the involved ECC-groups using some device-specific coding algorithms. The ECC "code" (7 or 8 bits) is stored with the data in memory. At a read to memory, or during memory scrubbing, the memory controller reads the involved ECC-groups, including both data and ECC codes. It also recomputes the ECC codes based on the data just read and compares it with the stored ECC codes. If they mismatch, the memory controller automatically corrects single-bit errors, and reports

multi-bit errors to the processor using an interrupt, which is delivered to the operating system.

To handle an ECC-error interrupt current operating systems, including both Linux and Microsoft Windows, simply go to the panic mode or the blue screen and report an error message to the end-user. The user has to reboot the machine to solve the problem.

### 3.2.2 Using ECC to Monitor Memory Accesses

#### Main Idea

SafeMem makes a novel use of ECC memory to monitor memory accesses for software debugging. More specifically, it leverages ECC memory for two purposes: (1) detecting illegal accesses (e.g., out-of-bound memory accesses, or accesses to freed memory buffers) to monitored memory locations; (2) pruning false positives in memory leak detection. More details about each specific usage are described in Section 3.3 and Section 3.4.

Both usages require detection of accesses to some monitored memory locations. To achieve this goal, SafeMem uses ECC protection in a way similar to page protection, which is commonly exploited in shared virtual memory systems [Li88]. Even though ECC groups are either 32 bits or 64 bits in granularity, using ECC for memory protection has to be at cache-line granularity, because accesses to main memory use this granularity.

The advantage of using ECC protection over using page protection is that the former is at cache line granularity, whereas the latter is at page granularity. Therefore, ECC protection can significantly reduce the amount of false sharing and padding space. The experiments compared these two approaches quantitatively, and the results show that ECC protection can reduce the amount of memory waste used for memory monitoring by up to 74 times (see Section 3.6).

These advantages of ECC protection are also exploited by some fine-grained distributed shared memory systems, such as Blizzard [SFL+94]. Different from those works, SafeMem leverages ECC protection for software debugging instead of implementing cache coherence operations. Therefore, SafeMem has different design trade-offs. In addition, they used special ECC mem-

ory controllers, whereas SafeMem uses a standard off-the-shelf ECC memory controller, which has much more limited functionality available to software. For example, most commercial ECC memory controllers do not allow software to directly access the ECC code. Moreover, unlike page protection faults, operating systems do not deliver the ECC-error interrupt to user-level programs. Therefore, it is the first step to address all these challenges before SafeMem uses ECC for monitoring memory accesses to watched locations.

I modify the Linux operating system to provide three new system calls: (1) *WatchMemory(address, size)*, which registers a memory region starting from *address* to be monitored by SafeMem. The memory region and its size need to be cache line aligned. (2) *DisableWatchMemory(address)*, which removes monitoring to the specified memory region. (3) *RegisterECCFaultHandler(function)*, which registers a user-level ECC fault handler. When an ECC fault occurs, the fault is delivered to this user-level handler.

SafeMem only needs to detect the first access to each monitored location because: (1) For memory corruption detection, the first access to a monitored location is a bug. SafeMem then simply pauses program execution to allow programmers to attach an interactive debugger, such as gdb, to check the program state and analyze the bug. (2) For memory leak detection, the first access to a monitored location indicates a false positive. Then this location no longer needs to be monitored. Therefore, in both cases, the user-level ECC fault handler of SafeMem can disable the monitoring for the faulted lines using *DisableWatchMemory()* system call.

### Design Issues

**Data Scrambling** Since most commercial ECC memory controllers do not allow software to directly modify an ECC code, SafeMem uses a special trick to "scramble" the ECC code of a watched ECC-group. When *WatchMemory* is called, SafeMem first disables the ECC functionality, and writes the scrambled data into this ECC-group. It then flushes the data from cache into memory. Since ECC is disabled, the ECC code for this line remains the same, i.e., the old code. Finally, SafeMem enables ECC. Figure 3.2 shows the process of this trick. During the disable-

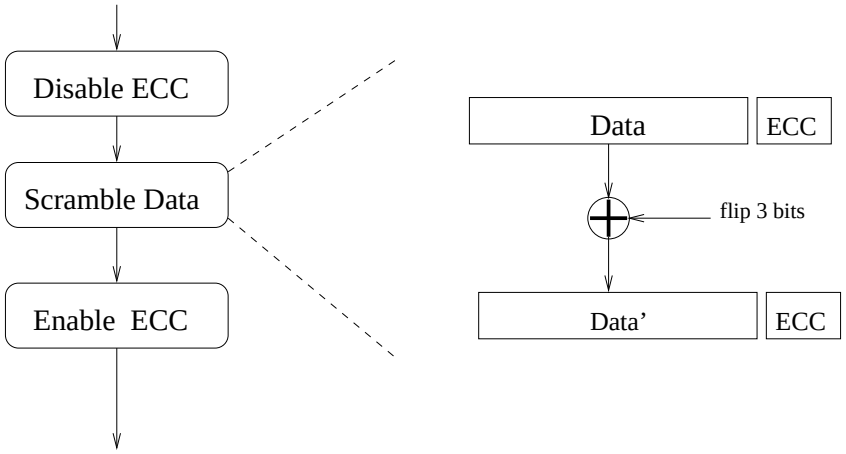



Figure 3.2: Implementation of *WatchMemory*

enable period, SafeMem locks the memory bus to avoid any other background memory accesses, such as those made by other processors or DMAs, so that other memory locations are not affected by this *WatchMemory* operation. After this operation, the first access to this location triggers an ECC fault because of the mismatch between the old ECC code and the scrambled data.

The data is not scrambled randomly. Instead, SafeMem uses a special scrambling scheme to ensure two properties: (1) The scrambled data should trigger a multi-bit ECC fault instead of a single-bit error, as most ECC memory can automatically correct single bit errors without reporting to the operating system. (2) The scrambled data should have a unique signature so that it can be easily differentiated from a real hardware ECC error. In the prototype implementation, SafeMem flips 3 fixed bits of the original data stored in a watched line.

In addition, SafeMem stores the original data in a private memory region of SafeMem in order to differentiate an access fault from a real hardware memory error. With the original data, the SafeMem ECC fault handler can recompute the "scrambled" value and compare against the current value stored in memory. If they do not match, it is a real hardware ECC error. Otherwise, it is an access fault caused by an access to this watched location.

**Differentiate Hardware Errors from Access Faults** The main functionality of ECC memory is to detect memory hardware errors, which does not interfere with SafeMem for two reasons.

First, as mentioned earlier, SafeMem scrambles data in a special way. When an ECC fault occurs, SafeMem first checks whether the line is monitored; if so, SafeMem checks the data to see whether it matches the scrambling signature. If yes, it is an access fault, otherwise, it is a hardware error. Second, the data stored in monitored regions is not useful because monitored regions are either padded ends or leaked buffers. Therefore, even if the data is modified because of a real hardware error, it is not critical to the program's execution. Moreover, the original data in monitored regions is saved in SafeMem's private memory.

**Dealing with ECC Memory Scrubbing** When the memory controller enables scrubbing, memory is scanned periodically to check and correct hardware errors. Therefore, special care needs to be taken in order to avoid undesired ECC faults introduced by memory scrubbing. Since most ECC memory controllers allow the OS to dynamically enable/disable scrubbing, SafeMem solves this problem by coordinating with ECC memory controllers in the following way: during scrubbing, SafeMem temporally un-monitors all the watched regions and blocks the monitored program until scrubbing finishes. Since scrubbing is infrequently performed and only during idle periods, this will not significantly affect performance. However, a better alternative would be to scrub and unmonitor the memory at page granularity, which would require changes to ECC memory controllers to signal the OS before each page scrubbing.

**Dealing with Cache Effects** To avoid the cache filtering effect, the *WatchMemory* operation flushes the corresponding cache line from the processor caches so that subsequent accesses to this line must access memory and therefore trigger the corresponding ECC fault. This technique also ensures that a write instruction to a watched line is also monitored (even though writes to memory do not trigger ECC checks). This is because a write to data that is not currently in cache must first load the data from memory to cache, and thereby triggers an ECC fault. After the first access is detected, the line can remain in the processor cache without being flushed because SafeMem only needs to detect the first access to a watched line.

**Dealing with Page Swapping** Since ECC protection is associated with physical memory, it can be affected by page swapping which changes the virtual-to-physical page mapping. A simple way to address this problem is to pin monitored pages: a page is pinned when any memory region inside is monitored, and is unpinned when it has no monitored memory regions. However, this method limits the total amount of monitored memory. To solve this problem, a better solution would be to modify the OS to un-monitor all associated memory regions when a page is swapped out, and re-monitor those regions when this page is swapped in. For simplicity, I implement the first method in SafeMem.

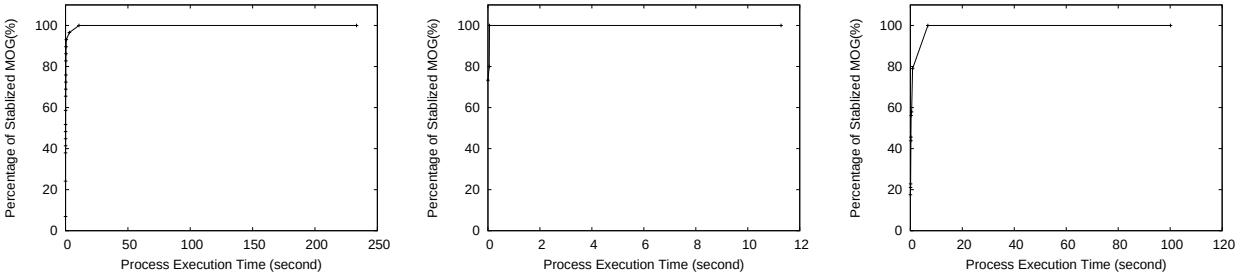
### Discussion

Unfortunately, ECC has several limitations that cannot be overcome by simply using software tricks. Addressing these limitations requires hardware changes. For example, even though ECC protection is much finer grained than page protection, it is still larger than desired. In SafeMem, each dynamic buffer requires padding space of two cache lines. In addition, each dynamic buffer size needs to be cache-line aligned to avoid false sharing, which also wastes memory space. If ECC protection could be done at word granularity, such as in the Mondrian Memory Protection (MMP) [WCA02], the amount of memory waste could be further reduced. Unfortunately, Mondrian Memory Protection still does not exist in real hardware yet.

Some aspects of the current ECC library in SafeMem are device-specific. The reason is that most ECC memory controllers export a narrow, limited interface to OS. Since this study provides a strong motivation to utilize ECC for purposes other than hardware memory error detection and correction, we hope that the ECC-protection interface can be generalized to be more software-friendly, just like page protection. In other words, the interface should include the following two features: (1) An ECC memory controller allows the OS to directly modify the ECC code associated with any data. This feature is not only useful for applications like SafeMem, but also allows software to dynamically fix some transient memory errors without going to panic mode. (2) An ECC memory controller can deliver precise interrupts of ECC faults to the OS so that the OS

can catch exactly the faulted instruction. Even though SafeMem does not need this feature for bug detection, this feature would allow SafeMem to enhance its functionality, such as providing programmers with precise information regarding the occurred bugs. With the above two features, SafeMem could be designed with a better hardware-software layered architecture.

## 3.3 Detecting Memory Leaks




(a) *ypserv* (b) *proftpd* (c) *squid*

Figure 3.3: Stability of maximal lifetime (MOG means Memory Object Group).

Not all memory leaks affect software reliability and availability. *Trivial memory leaks* (leaks that only happen several times) result only in memory waste and a slight execution slowdown due to increased paging. In contrast, *continuous memory leaks* (non-stop leaking) can cause programs to run out of virtual memory and eventually crash. Crashes are especially catastrophic for long-running server programs, such as web servers, because service unavailability is directly related to loss of business. Therefore, continuous leaks are often exploited by malicious users to launch denial of service attacks.

This chapter focuses on continuous leaks because they make software vulnerable. The detection method in SafeMem first analyzes the run-time dynamic memory usage behavior of a program, then uses the learned behavior to detect outliers, and finally exploits ECC-protection to prune false positives.

For the convenience of description, the following terminology is used throughout this chapter:

- *Memory Object:* a memory block allocated via memory allocation calls such as *malloc*,

*realloc, calloc,* etc.

- *Live Memory Object:* a memory object that is not yet deallocated.
- *Lifetime of Memory Object:* the period from the allocation of a memory object to its deallocation.
- *Memory Object Group:* a group of memory objects. In this chapter, I use a tuple $(size, callChain)$ to divide memory objects into various groups, where $size$ is the object's size, and $callChain$ is the call-stack signature[1] when the object is allocated. Even though it is possible to use other grouping methods, such as program-specific types, the experiments show that the grouping mechanism in SafeMem works well and does not require any semantic information from programs.

### 3.3.1 Characteristics and Classification of Continuous Memory Leaks

There are two main types of continuous memory leaks for a memory object group, and each type has different characteristics. The first type, called *always leak (ALeak)*, refers to leaks that always happen. In other words, the program does not free a group of memory objects in all possible execution paths. As a result, the number of memory objects in this group grows rapidly, and each object has an infinite lifetime. Detecting this type of memory leaks is relatively easy since it has simple characteristics.

The second type, called *sometimes leak (SLeak)*, refers to leaks that sometimes happen. In other words, in some execution paths, the program deallocates the allocated memory object, but in the other paths, the program does not free the allocated memory object. Therefore, some memory objects have finite lifetime whereas other objects in the same group have infinite lifetime. The number of leaked memory objects grows slowly, but it can still lead to memory resource exhaustion

[1] The call-stack signature is calculated by individually applying the exclusive-OR and rotate functions to the return addresses of the most recent four functions in the current stack.

after a long period of time, resulting in program crashes. The second type is much harder to detect since the leak happens only in some execution paths.

Fortunately, based on the memory usage behavior analysis using several server programs, I found that most dynamic memory objects conform to some expected lifetime. More specifically, the maximal lifetime of memory objects that belong to the same group usually remains stable after some warm-up period. Therefore, if SafeMem can dynamically capture the maximal lifetime for each object group, it can detect outliers—memory objects whose lifetime significantly exceeds the expected maximal lifetime of the corresponding object group.

Time here means the CPU time of the monitored program, which excludes time used by other running programs and time waiting for I/Os. Therefore, for server programs, *a long idle period between two consecutive client requests would not affect our detection mechanism*.

This observation is validated through statistical analysis using three server programs. To measure the stability of maximal lifetime for a memory object group, this chapter introduces a metric called *WarmUpTime*, which denotes how long it takes for this group's maximal lifetime to become stable. For a given memory object group, after the *WarmUpTime*, objects that belong to this group never live longer than this maximal lifetime.

Figure 3.3 shows the stability of maximal lifetime for three server programs: ypserv, proftpd, and squid, which are later used in the experiments to evaluate SafeMem. When collecting statistics, I use normal inputs so the memory leak bugs do not occur. Each curve on Figure 3.3 plots the cumulative distribution of memory object groups whose *WarmUpTime* is smaller than a given value. For example, a point $(t, p)$ on the curve indicates that $p$% of the memory object groups in this program have reached the stable maximal lifetime after running for $t$ seconds. Each memory object group is labeled by a tuple $(size, callChain)$, described in the previous subsection.

As shown in Figure 3.3, for all three programs, all memory object groups reach their stable maximal lifetime quickly in the very beginning of the program execution. I have also run the programs much longer, but the results remain the same. This validates our observation that the expected maximal lifetime remains stable after some short warm-up periods. Therefore, it can be

used to detect potential memory leaks by dynamically monitoring each memory object's lifetime against the expected maximal lifetime associated with the corresponding object group.

### 3.3.2 Detection Process

Based on the above observation, SafeMem detects these two types of continuous memory leaks on-the-fly during production runs. The detection process includes three steps: (1) Dynamically analyze the memory usage behavior of the monitored program; (2) Detect potential memory leaks based on observed usage characteristics; (3) Use ECC protection to prune false positives.

Each of the three steps adds only a small overhead because step 1 and step 2 are performed periodically and only at memory allocation or deallocation time instead of every memory access, and step 3 is performed only for those rare memory leak suspects. The first access to a suspect disables ECC monitoring for this memory object.

#### Step1: Memory Usage Behavior Collection

For each memory object group, SafeMem dynamically collects its allocation/deallocation behavior. More specifically, SafeMem records two types of information: (1) lifetime information and (2) memory usage information. The lifetime information includes the current maximal lifetime and how long the maximal lifetime has been stable (*stableTime*). Once again, time here is measured using the CPU time.

The memory usage information includes the number of current live objects, the last allocation time, and the total memory space currently occupied by this memory object group. For each live memory object, it also records its allocation time. All live objects within the same group are linked together using a double-linked list.

At each memory allocation, the information associated with the corresponding memory object group is updated. More specifically, a new live object is added to this memory object group, and the number of current live objects is incremented by one. The last allocation time and the total memory space currently occupied by this memory object group are also updated accordingly.

Similarly, the information is also updated at each memory deallocation. First, the lifetime of the deallocated object is calculated by subtracting the current time by its allocation time. If the lifetime is smaller than or within some tolerable range (based on a pre-defined threshold) from the maximal lifetime associated with the corresponding object group, the maximal lifetime remains unchanged, and its *stableTime* is incremented by the elapsed CPU processing time from the last update. Otherwise, the maximal life time is updated to be this object's lifetime and the *stableTime* is reset to zero. Finally, other information, such as the number of current live objects and the total memory space currently occupied by this memory object group, is also updated.

This step is implemented by wrapping the memory allocation/deallocation functions such as *malloc(), calloc(), realloc(), free()*, etc. For programs that use their own memory allocators, SafeMem needs to wrap their allocation and free functions.

**Step2: Outlier Detection**

The detection techniques are different for different types of memory leak. For each memory object group, it first checks whether this group has ever called deallocation before. If so, it follows the detection procedure for SLeaks (sometimes-leak). Otherwise, it continues the process of ALeak (always-leak) detection.

To detect ALeaks, SafeMem monitors the memory usage behavior. It first checks whether the number of live objects of each object group exceeds some given threshold. If so, it then checks whether the memory usage by this group is continuously growing. This is done by checking the last allocation time associated with this group. If the last allocation time is long time ago (compared to the current time), the memory usage is not dynamically growing. This is unlikely to be memory leaks. Instead, it might be the case that the program allocates many objects at initialization time and these objects are used throughout the entire execution. However, if the last allocation time is very recent, it indicates that the memory usage is still growing. Therefore, this group of memory objects are leak suspects, which should be monitored using ECC protection for false positive pruning.

To detect SLeaks, SafeMem monitors the lifetime of each live object. An object is singled

out as a suspect to be monitored using ECC protection if two conditions hold: (1) this object has been alive for more than two times its expected maximal lifetime, and (2) the maximal lifetime for the corresponding object group has been relatively stable for a period of time (longer than a given threshold). If condition 2 is not true, no outliers will be singled out because the detection confidence is very low in such cases. Because all live memory objects of the same group are linked in the order of their allocation time, SafeMem only needs to check the top few oldest memory objects' lifetimes to detect potential SLeaks.

The detection process is triggered after a warm-up period, and is periodically performed only at memory allocation/deallocation time. More specifically, at each memory allocation/deallocation, if the elapsed time from the last check is greater than a pre-defined parameter, called the *checking-period*, the detection process is performed. Therefore, this step has a very small overhead.

It is safe to perform the detection process only at memory allocation/deallocation time. If the program has not performed any allocation/deallocation for a long time, there is no need to trigger the detection process because the memory usage is not actively growing. Therefore, even if some memory objects have already been leaked, it will not cause the program to crash since the memory usage has stopped growing. As mentioned before, our study focuses on detecting continuous memory leaks that can affect system reliability and availability.

**Step3: False Positive Pruning Using ECC Protection**

When an object is marked as a suspect during Step 2, it is monitored using ECC protection to prune false positives from real leaks. This is based on the observation that if a suspect is accessed again, it is unlikely to be a memory leak. If it has never been accessed for a threshold of time, it is reported as a memory leak.

The pruning procedure works as follows. Each suspect is monitored by calling *WatchMemory*. The first access to this suspect will trigger the ECC protection handler which then removes this object from the suspect list and turns off the ECC monitoring for this object. If this suspect is an SLeak suspect, this object's allocation time is reset to the current time to catch possible future

leaks (an object can become a suspect again if it continues to live longer than the expected maximal lifetime). The maximal lifetime associated with this object group is then updated to be the current living time of this suspect to avoid other similar false positives.

The pruning process does not impose significant overhead since it is only performed on rare suspects. In addition, only the first access to a suspect needs to pay the extra overhead of triggering and executing the ECC fault handler.

## 3.4 Detecting Memory Corruptions

Memory corruption can be caused by many reasons, among which buffer overflow and accesses to freed memory are two of the most common. Buffer overflow is a particularly important type of memory corruption because it is often exploited by viruses to attach and execute malicious code. Therefore, SafeMem focuses on detecting buffer overflows and accesses to freed memory, both of which are also the major types of bugs detected by Purify.

To detect buffer overflow, SafeMem pads the two ends of each buffer and then uses ECC protection to guard these paddings; any accesses to the padding are reported as buffer overflow bugs. The current implementation of SafeMem uses a cache line as the padding unit. It could easily use longer paddings, but the experiments on applications with buffer overflow bugs show that the current setting is good enough. To reduce false sharing, each memory buffer is cache line aligned. When a buffer is deallocated, the ECC monitoring of its paddings is disabled.

To detect accesses to freed memory, SafeMem uses ECC protection to watch all freed memory buffers. An access to such a buffer will trigger the ECC fault handler which reports this access as a bug. When a freed memory buffer is reallocated, ECC monitoring for this buffer will be disabled. Similar to buffer overflow detection, each memory buffer and its size need to be cache line-aligned to avoid false sharing.

The overhead to detect both types of memory corruption is relatively small because it only needs an extra system call at the memory allocation/deallocation time. Since most programs do

not have very frequent allocation/deallocation, the overhead imposed by SafeMem is small, as shown in our experimental results (See Section 3.6).

ECC protection can also be used to detect other types of bugs, even though the current implementation of SafeMem does not support them yet. For example, accesses to uninitialized objects could also be detected using ECC protection. After a memory buffer is allocated, it can be protected using ECC protection. The first write to this buffer would disable the ECC protection, but the first read would be detected and reported as a bug.

## 3.5 Methodology

### 3.5.1 Platform

The experiments are conducted on a real system with a 2.4 GHz Pentium processor, an ECC memory controller with the Intel E7500 chipset [Int], and 1 GByte of memory. The operating system extensions (the three new system calls) are added into Linux kernel 2.4.20. SafeMem is implemented as a shared library and can be dynamically preloaded in advance to avoid recompilation of the tested programs (unless the programs use their own memory allocators, in which case we need to do some simple changes to intercept their memory allocation/deallocation calls).

The evaluation compares the time overhead of SafeMem to Purify [HJ92b], a state-of-the-art dynamic bug detection tool. Purify can detect memory corruption and memory leak bugs. More specifically, in order to find memory-access errors, Purify maintains two bits for each byte of memory to track its status: allocated or freed, and initialized or uninitialized. Purify checks each memory operation against its status and reports illegal accesses. As for memory leaks, at some point during program execution or when the tested program exits, Purify applies an algorithm similar to the conventional mark-and-sweep algorithm [HJ92b], which utilizes conservative pointer tracking to scan the whole heap. Performing such an expensive operation adds large overhead and also significantly perturbs the program's response time, especially for server programs. Therefore,

these tools are always used for in-house debugging instead of during production runs.

We evaluate seven different real-world, buggy applications shown on the Table 3.1, from complicated network server daemons, such as squid and proftpd, to simple common utilities, e.g., gzip. These tested applications are divided into two groups: one containing memory leaks, and the other containing memory corruption bugs.

Based on these applications, we have conducted two sets of experiments. The first set evaluates the functionality of SafeMem in detecting bugs, and the second set compares the overhead of SafeMem to Purify's using bug-free runs of the tested applications (with normal inputs). We also evaluate the benefits of ECC protection in reducing memory waste and pruning false positives.

| Bugs | Application | LOC | Description |
|---|---|---|---|
| Memory Leak | ypserv1 | 11,200 | a NIS server |
| | proftpd | 68,700 | a ftp server |
| | squid1 | 95,000 | a Web proxy cache server |
| | ypserv2 | 9,700 | a NIS server |
| Memory Corruption | gzip | 8,900 | a compression utility |
| | tar | 34,000 | an archiving utility |
| | squid2 | 93,000 | a Web proxy cache server |

Table 3.1: Tested applications (LOC means lines of code). squid1 and squid2 are different versions of squid, but one contains memory leaks and the other contains a memory corruption bug. Similarly, ypserv1 and ypserv2 are different versions of ypserv, but one contains ALeaks and the other contains SLeaks.

Even though several studies [NMW02] have directly compared their tools with Purify for detecting only one type of bug, memory corruption, we do note that Purify can check for other types of bugs, such as accesses to uninitialized variables, which are not detected by SafeMem. Unfortunately, the current version of Purify does not provide options to allow disabling these checks to make the comparison fair. However, based on our experience and understanding of Purify's techniques, disabling these checks would not reduce its overhead significantly. After all, Purify needs to monitor every memory access no matter whether it is for detecting memory corruption or for detecting accesses to uninitialized variables. Moreover, this does not have much impact on the interpretation of our results since SafeMem has a substantial overhead reduction over Purify.

## 3.6 Results

### 3.6.1 Microbenchmark Results

First, I conduct some microbenchmarks to measure the cost of the ECC monitoring system calls. Table 3.2 shows the cost for the *WatchMemory()* and *DisableWatchMemory()* system calls on our machine. The costs for these two calls are relative cheap (less than 2 microseconds), comparable to the page protection call *mprotect()* provided by the standard Linux system. Ours are slightly higher than *mprotect* because our calls need to pin (unpin) the page in the virtual memory system.

### 3.6.2 Overall Results

Table 3.3 shows the overall results of SafeMem with seven buggy applications. First, SafeMem can detect all the tested bugs (both memory leaks and memory corruption). This shows that SafeMem is effective in achieving its expected functionality.

I also compare SafeMem's overhead with Purify's. For fair comparison, *SafeMem enables both memory leak detection and memory corruption detection for all experiments, even though each application has only one type of bug.* To avoid disturbance by the bugs, I use normal inputs when measuring overheads so the bugs do not occur and program can run correctly to completion.

As shown in Table 3.3 (column "ML+MC"), SafeMem adds only 1.6%-14.4% overhead for all tested applications, a factor of 114-1660 times smaller than Purify's overhead (4.8X - 49.8X). For example, for gzip SafeMem adds only 3.0% overhead, whereas Purify slows down this application by a factor of 49.8. This is because SafeMem does not need to monitor each memory

| | Calls | Time(microseconds) |
|---|---|---|
| ECC Protection | WatchMemory | 2.0 |
| | DisableWatchMemory | 1.5 |
| Page Protection | mprotect | 1.02 |

Table 3.2: Time for the ECC system calls

| Bugs | Apps | Bug Detected? | SafeMem Overhead(%) of Detecting | | | Purify Overhead (%) | Reduction by SafeMem |
|---|---|---|---|---|---|---|---|
| | | | Only ML | Only MC | ML + MC | | |
| Memory Leak (ML) | ypserv1 | YES | 1.0 | 4.2 | 6.0 | 941 | 157X |
| | proftpd | YES | 0.9 | 2.6 | 3.6 | 2093 | 581X |
| | squid1 | YES | 5.6 | 7.8 | 13.7 | 1782 | 130X |
| | ypserv2 | YES | 0.7 | 10.5 | 11.5 | 1308 | 114X |
| Memory Corruption (MC) | gzip | YES | 0.3 | 2.2 | 3.0 | 4979 | 1660X |
| | tar | YES | 0.7 | 1.0 | 1.6 | 475 | 297X |
| | squid2 | YES | 6.1 | 8.1 | 14.4 | 1720 | 119X |

Table 3.3: Time overhead (%) comparison between SafeMem and Purify

access. Instead, it relies on ECC protection and intelligent memory usage behavior analysis to detect memory corruption and memory leaks. In contrast, Purify needs to intercept every memory access in order to detect memory corruption, and needs to do a mark-and-sweep over the entire memory space in order to detect memory leaks. The small overhead indicates that SafeMem can be used to detect memory leaks and memory corruption *during production runs*.

I further measure SafeMem's overhead for detecting only memory leaks and detecting only memory corruption, respectively. The memory leak detection overhead comes mainly from the information collection and analysis, whereas the memory corruption overhead comes mainly from the ECC monitoring and un-monitoring. Table 3.3 also shows that overhead caused by memory corruption detection is more than that caused by memory leak detection. This is because memory corruption detection needs to enable ECC monitoring at each buffer allocation and disable ECC monitoring at each deallocation. Memory leak detection, however, only enables monitoring for the suspected memory objects, which usually is many fewer than the total number of allocated memory objects.

### 3.6.3 Benefits of ECC Protection

Table 3.4 shows the benefit of ECC protection over page protection in reducing memory waste for padding and alignment. As shown on this table, ECC-protection adds only 0.084%-334% of total memory overhead (not necessarily used at the same time) for the tested applications, whereas page-

| Bugs | Application | Memory Overhead(%) | | Reduction by ECC |
|---|---|---|---|---|
| | | ECC- | Page-Protection | |
| Memory Leak | ypserv1 | 57 | 3900 | 68X |
| | proftpd | 35 | 2357 | 67X |
| | squid1 | 26.4 | 1950 | 74X |
| | ypserv2 | 3.6 | 233 | 64X |
| Memory Corruption | gzip | 0.084 | 6.06 | 72X |
| | tar | 334 | 23178 | 69X |
| | squid2 | 28.7 | 2120 | 73X |

Table 3.4: Space overhead (%) comparison between ECC- and page-protection approaches. The overhead is calculated over each applications' actual memory usage throughout the whole execution.

| Application | False Positives | |
|---|---|---|
| | Before Pruning | After Pruning |
| ypserv1 | 7 | 0 |
| proftpd | 9 | 0 |
| squid1 | 13 | 1 |
| ypserv2 | 2 | 0 |

Table 3.5: False memory leaks reported before and after using ECC-protection. No false positives for memory corruption detection by SafeMem.

protection has 6.06%-231.78X of memory space overhead! In other words, ECC-protection can reduce the memory waste of page-protection by a factor of 64-74! This shows that ECC protection is a better mechanism to use for detecting memory leaks and memory corruption.

### 3.6.4 Effects of ECC-Protection in False Pruning for Memory Leaks

Table 3.5 reports the effects of ECC-protection in false pruning for memory leaks. The results show that this pruning mechanism is very effective: it is able to reduce the number of false positives from 2-13 to 0-1. For example, for squid1, without this pruning scheme, SafeMem would have introduced 13 false positives instead of 1 false positive, which is much harder for programmers. SafeMem does not have any false positives in memory corruption detection because any accesses to padding areas or freed memory buffers are true memory corruption.

## 3.7 Summary

This chapter presents an approach called SafeMem that makes a novel use of ECC memory for detecting memory leaks and memory corruption, two major forms of software bugs that contribute significantly toward software vulnerabilities. SafeMem does not require any new hardware extensions and can work with existing systems with ECC memory, which is commonly used in modern systems. Moreover, this chapter also present a new method that uses intelligent memory usage behavior analysis to detect memory leaks.

This chapter evaluated SafeMem using seven real-world buggy applications and the result indicates that SafeMem can be used for building a first-level defense for detecting memory leaks and memory corruption during production runs. The results show that SafeMem can detect all tested bugs with only 1.6%-14.4% overhead, 2-3 orders of magnitude smaller than the commonly used commercial tool, Purify. Moreover, the results also show that ECC protection can reduce the amount of wasted memory by a factor of 64-74 compared to page protection. Finally, ECC protection is also very effective in pruning false positives for memory leak detection.

# Chapter 4

# Second-Level Defense: Detecting Bug Exploits

## 4.1 Overview

### 4.1.1 Motivation

As discussed in Chapter 1, during production runs not all triggered software bugs can be caught in the first-level defense. Those escaped bugs such as stack buffer overflow and double free may be exploited by malicious users for launching security attacks. The question is how we should address them to control the system damage that may be caused by those attacks. This chapter proposed a low-overhead, software-only information flow tracking system, called LIFT, to build the second-level defense. LIFT minimizes the runtime overhead by exploiting dynamic binary translation and optimizations for detecting various types of security attacks without requiring any hardware changes.

As reviewed in Chapter 2, several recent work [CCC+05, NS05, SLZD04] demonstrated that information flow tracking is a promising and effective technique to effectively detect many system-compromising security attacks, even for *unknown* types of exploits and software vulnerabilities. Generally this technique tags (labels) the input data from unsafe channels such as network connections as "unsafe" data, propagates the data tags through the computation (any data derived from unsafe data are also tagged as unsafe), and detects unexpected usages of the unsafe data that switch the program control to the unsafe data as exemplified in the stack smashing attack. In addition to the generality of attack detection, information flow tracking can also trace back to the input data that exploits the vulnerability to generate attack input signatures. This feature has been demon-

strated to be very useful for effectively building a preventive network defense [CCC+05, NS05].

Table 4.1 shows an example to demonstrate the information flow tracking process. Initially, $a$ is received from the network, so it is unsafe. The second statement makes the information of $a$ flowing to $b$. When the program jumps to the location pointed by $c$, the system will raise an alarm if $c$ is unsafe.

| Target Program | Information Flow Tracking |
|---|---|
| receive (&a); | $Tag(a) = 1$ // unsafe as it is received from network |
| b=a; | $Tag(b) = Tag(a)$ |
| ... | ... |
| jmp c; | if $(Tag(c) == 1)$, raise alert! |

Table 4.1: An example of information flow tracking

So far information flow tracking has been implemented in three different ways. The first approach is to track information flow at compile time for programs written in special type-safe programming languages [Den76, DD77, HR98, Mye99, ML00]. While this approach can enforce the information flow security policies for programs without runtime overhead, it only works for programs that are written in the specific program languages and is therefore inapplicable to a large number of legacy programs written in type-unsafe languages such as C/C++. More importantly, due to lack of accurate runtime information, most tools in this category are designed for detecting sensitive information leaking instead of security attacks due to lack of accurate runtime information. For example, it is hard for them to detect attacks that alter the target of indirect branches, which can only be resolved at runtime.

The second approach is to track information flow and detect malicious exploits at runtime via either source code or binary code instrumentation. Source-code instrumentation-based information flow tracking, as done in Xu et al's work [XBS06], has lower overhead than the alternative, binary-code instrumentation-based implementation, but it cannot track information flow in third-party library code and thereby will miss security exploits involving these libraries as reported in US-CERT [US-06]. Additionally, it requires programmers to provide a summary for each library

function to allow the information flow through library calls, which can be an error-prone and tedious task as many library calls are fairly complex, making many side-effects in addition to simple return values. In contrast, implementing information flow tracking via binary instrumentation as Vigilante [CCC+05] does can track information accurately even in libraries, but suffers from a major overhead problem: it can slow down the program execution by more than 40 times [NS05], too large to be used during production runs against security attacks.

The third approach of information flow tracking is to support it in hardware [SLZD04, VBC+04]. For example, the recent work RIFLE [VBC+04] and Suh et al's work [SLZD04] proposed new hardware extensions to track information flow for each instruction. While this approach is effective in detecting security attacks with low overhead, it requires non-trivial hardware extensions. Therefore, it is quite expensive and is not applicable to existing systems.

### 4.1.2 Highlight of LIFT

This chapter proposes a *low overhead*, *software-only*, *comprehensive* and practical information flow tracking system, called LIFT. LIFT minimizes run-time overhead by exploiting *aggressive dynamic binary instrumentation and optimizations* for detecting various types of security attacks *without requiring any hardware changes*. Dynamic binary instrumentation and optimizations leverage accurate runtime information and enable more aggressive optimizations than static approaches at compile time. For example, at runtime we can eliminate many unnecessary *dynamic* information flow tracking for execution periods when it can be sure that there is no unsafe data involved in the computation.

More specifically, LIFT employs three runtime binary optimizations to minimize the overhead associated with information flow tracking for detecting general security attacks. The first optimization, referred as Fast Path (FP), eliminates unnecessary dynamic information flow tracking. This is based on the observation that, for most applications, the majority of computation involves safe data, for which it is unnecessary to track information flow. Therefore, by dynamically and efficiently performing a simple check before an execution region (e.g. basic block), LIFT can see

whether involved data is safe or not; if it is, a fast binary version without any information flow tracking is executed; otherwise, the execution follows a slow version with detailed information flow tracking. By dynamically switching between fast and slow versions on demand, LIFT can effectively avoid unnecessary information tracking.

The second optimization, Merged Check(MC), further reduces the information flow checking overhead by coalescing data safety checks from multiple consecutive basic blocks into one. This optimization exploits both spatial locality and temporal locality of memory references because multiple safety checks for both nearby data and the same data are combined into one. It not only reduces the number of checks but also avoids bit operations because the safety of one byte is indicated by only one bit in the corresponding data tag. This optimization is applied to both consecutive basic blocks but also dynamic instruction traces (i.e. dynamically-formed frequently executed code regions).

The last optimization, Fast Switch (FS), reduces the overhead and number of context switches [1] between the target program and the information flow tracking code by using alternative cheaper instructions and status register liveness analysis, respectively. To avoid interference with the target program, most binary instrumentation frameworks such as PIN [LCM+05] and StarDBT [BWWA06] usually require saving/restoring some program execution context, including the status register and the runtime stack pointers, of the target program before and after executing the instrumented code (the reason will be discussed in detailed in Section 4.3.3). This context switch, even though much smaller than OS-level context switches, can still introduce large runtime overhead. LIFT minimizes this overhead by cleverly selecting cheaper instructions and performing register liveness analysis.

I have implemented LIFT based on a dynamic binary translator called StarDBT [BWWA06] on Windows. The *real-system* experiments with two real-world *server* applications, one client application, and *eighteen* attack benchmarks [WK03] show that LIFT can effectively detect various types of security attacks. LIFT also incurs very low overhead, only 6.2% for server applications,

[1] Note that the context switch here is *not* the OS-level context switch between different threads or kernel-user mode.

and 3.6 times on average for seven SPEC INT2000 applications. The dynamic optimizations in LIFT are very Effective in reducing the overhead by a factor of 5-12 times.

Compared to previous approaches, LIFT provides several unique advantages:

- *Low-overhead.* Compared to other software-only binary-based information flow tracking system that slows down program execution by more than 40 times, LIFT incurs significantly less overhead, only 6.2% for server applications, and 3.6 times on average for seven SPEC INT2000 applications, which indicate that LIFT is practical to use during production runs for detecting security attacks.

- *Not requiring any hardware extensions.* LIFT is a software-only approach based on dynamic binary instrumentation and optimization. Unlike previous hardware-based approaches [VBC+04, SLZD04] that requires non-trivial hardware extensions, LIFT requires no hardware extension. Therefore, it can be used immediately in existing systems.

- *Not requiring source code.* Unlike source-level information flow tracking [XBS06], LIFT works with binary code and thereby can work with commercial software whose source code is unavailable. In addition and most importantly, it can performs accurate information flow tracking inside third-party library code and, consequently, can detect security vulnerabilities and exploits that occur inside these libraries.

The rest of the chapter is organized as follows. Section 4.2 describes the design and implementation of the basic information flow tracking system (LIFT-basic), followed by the three optimization techniques described in Section 4.3. Then Section 4.4 and Section 4.5 evaluate LIFT, followed by summary in Section 4.6.

## 4.2 LIFT Basic Design and Implementation

LIFT tracks information flow at runtime via dynamic binary translation and optimization to detect general security attacks. Similar to other information flow tracking systems [SLZD04, NS05,

CCC+05], LIFT dynamically instruments the binary of the target program to perform two tasks: (1) tracking information flow, and (2) detecting security exploits that switch the program control flow to unsafe data.

This section describes the basic design and implementation including the basic dynamic binary instrumentation framework, tag management, information flow tracking, exploit detection, an example of information flow tracking for instructions, and protection of tag space and LIFT code. The three dynamic optimizations for minimizing runtime overhead will be described in the next section.

### 4.2.1 Dynamic Binary Instrumentation Framework

I build LIFT on top of a dynamic binary translator called StarDBT [BWWA06] developed by Intel. StarDBT automatically loads the original program code into memory and initializes the program execution context at program startup time. Like other dynamic binary instrumentation and translation frameworks [BDB00, SBB+03, LCM+05], StarDBT manages a code cache to store the translated code so that the original code is translated once and executed multiple times in order to amortize the translation cost. In addition, StarDBT collects profiling information to form hot traces of frequently executed code. More details about the basic dynamic binary translation and instrumentation framework can be found in [BWWA06].

At run time, LIFT uses StarDBT to instrument the translated code with instructions that perform information flow tracking and attack detection. Besides StarDBT, LIFT can also be built on top of other dynamic binary instrumentation tools or translators such as Dynamo [BDB00], PIN [LCM+05], etc.

### 4.2.2 Tag Management

Similar to prior work [SLZD04], LIFT associates a one-bit tag (0 for “safe” data and 1 for “unsafe” data) for each byte of data in memory or general data registers. It can be easily extended to

a multiple-bit tag for each byte as needed. For example, users may want to use different tags to express their trustiness for data from different sources such as network data, disk data, and other data. Using multiple-bit tags can also reduce some overhead by avoiding bit operations in information flow tracking as demonstrated in prior work [XBS06], but it significantly increases the space overhead for keeping tags and also increases processor cache pollution. Therefore, the current prototype of LIFT uses one-bit tags.

LIFT stores the tags for memory data in a special memory region, called the *tag space*, via a one-to-one direct mapping between a tag bit and a memory byte in the target program's virtual address space. Such direct mapping makes it straightforward and fast (with only one memory access and 2-3 arithmetic instructions) to get the tag value for a given memory location.

The current tag space incurs 12.5% space overhead. If the virtual memory space is limited, we can minimize the tag space using compression as memory data nearby each other usually have similar tag values: either all zeros or all ones. So we may keep only one value for the entire memory region (e.g. a page). Although this scheme saves memory space, it has extra runtime overhead for tag look-ups. Since the current prototype of LIFT is based on 64 bits architectures, where virtual memory space is seldom limited, I only use the flat tag space management without any compression.

LIFT stores the tags for general registers in a dedicated extra register to minimize overhead. Since register accesses are very frequent in program execution, the register tags are also accessed frequently. Therefore, for efficient register tag accesses, LIFT uses an extra 64-bit register to store the tags for all registers used in the target program.

At the beginning, all tags are cleared to zero. Based on the application-specific tagging policy, certain data (e.g. data read from the network or standard input) are tagged with 1 as "unsafe". As the program continues, other data may also be tagged with 1 via information flow. An unsafe data can become safe if its value is reassigned from some safe data. All constants are safe data.

### 4.2.3 Information Flow Tracking

As program executes, LIFT propagates the tag information from one data to another. It does this by dynamically instrumenting instructions with information flow track according to its type. For data movement-based instructions such as MOV, PUSH, POP, etc, the tag value of the source operand is propagated to the tag of the destination (e.g, if the source operand is unsafe, the destination also becomes unsafe). For arithmetic instructions, such as ADD, OR, etc, the corresponding tag values of the two source operands are OR-ed and the result is propagated to the tag of the destination operand since the information of the destination operand comes from both source operands. For instructions that involve only one operand, such as INC, etc, the tag of the operand does not change since the information of the operand flows to itself. Similar to many previous work [XBS06, CCC+05, SLZD04, VBC+04], LIFT tracks information flows based on data dependencies but not control dependencies.

There are a few special instructions whose information flow tracking in LIFT does not follow the above general rules. For example, in x86 architecture, the instruction "XOR eax, eax" initializes the "eax" register to 0, therefore the tag value of "eax" should be reset to 0 ("safe" data). However, the general rule for this instruction keeps the tag of "eax" unchanged. To handle such cases, LIFT identifies these special instructions such as "XOR reg, reg" and "SUB reg, reg", and clear the tags of the corresponding registers or memory data.

In the baseline case (without any optimization described in the next section), the information flow tracking code is instrumented once at runtime and executed multiple times. The reason for instrumenting before instead of after an instruction in the original program is that execution of the instruction may change the operand address and thus make tag propagation more difficult.

### 4.2.4 Exploit Detection

In addition to information flow tracking, certain instructions are also instrumented to detect malicious exploits, i.e. improper usages of unsafe data that violate user-specified security policies. For

example, “unsafe” data cannot be used as a return address or the destination of an indirect jump instruction, etc.

By default, similar to previous work [CCC+05, SLZD04, NS05], LIFT detects any general security attacks, regardless of the underlying security vulnerabilities, which use “unsafe” data for jump targets, return addresses, function pointers, or function pointer offsets. This allows us to detect a wide variety of security attacks since the last step of most security attacks requires directly or indirectly changing the program control flow to some unsafe data by altering the return address, function pointers, or general jump targets.

### 4.2.5 An Example of Information Flow Tracking for LIFT-basic

Figure 4.1 shows an example of information flow tracking instructions for three instructions (with bold font) from a target program. For different instruction type, the number of instrumented instructions for information flow tracking varies. For example, the first instruction moves a constant to a register, whose information flow tracking takes eight instructions, while the second and third instructions from the target program each requires twenty instructions for information flow tracking or exploit detection respectively.

We use the second instruction “ADD ebx, [ecx]” from the target program as an example to see how the information flows. Instructions 1-5 do context switch, including switching to a different stack and saving the conditional flag register. Instructions 6-16 get the tag of the memory data “[ecx]”. Instructions 17-18 propagate the tag of source operand in memory to the tag of destination operand in the register. The last two instructions restore the context.

### 4.2.6 Protection of Tag Space and LIFT Code.

In addition to overhead, another important concern is that LIFT code or the tag space can be corrupted by some program errors or carefully-crafted malicious inputs. Therefore, it is necessary to protect them. To protect the LIFT code against corruption, I use page protection to make the

| | | |
|---|---|---|
| | MOV r10, gs:[30h]<br>MOV r10, [r10+1488h]<br>MOV [r10−8], rsp<br>LEA rsp, [r10−8]<br>PUSHFQ<br>XOR r11, r11<br>LEA r11d, [ecx]<br>MOV r10d, r11d<br>SHR r11d, 3<br>ADD r11, Tag_Space_Base<br>MOV r13, [r11]<br>AND r10d, 0x07h<br>XCHG r10d, ecx<br>SHR r13, cl<br>XCHG r10d, ecx<br>AND r13, 0x0fh<br>SHL r13, 0x04h<br>OR RegTag, r13<br>POPFQ<br>POP rsp | MOV r10, gs:[30h]<br>MOV r10, [r10+1488h]<br>MOV [r10−8], rsp<br>LEA rsp, [r10−8]<br>PUSHFQ<br>XOR r11, r11<br>MOV r11d, ebx<br>MOV r10d, r11d<br>SHR r11d, 3<br>ADD r11, Tag_Space_Base<br>MOV r13, [r11]<br>AND r10d, 0x07h<br>XCHG r10d, ecx<br>SHR r13, cl<br>XCHG r10d, ecx<br>AND r13, 0x0fh<br>TEST r13, 0x0Fh<br>JNZ report_intrusion<br>POPFQ<br>POP rsp |
| MOV r10, gs:[30h]<br>MOV r10, [r10+1488h]<br>MOV [r10−8], rsp<br>LEA rsp, [r10−8]<br>PUSHFQ<br>AND RegTag, 0xffffff0fh<br>POPFQ<br>POP rsp<br>**MOV ebx, 0x0400h** | **ADD ebx, [ecx]** | **JMP ebx** |

Figure 4.1: Information flow tracking for three different instructions in LIFT-basic. The instruction with bold font is an original instruction from the target program. The non-bold instructions instrumented *before* the bold instruction perform information flow tracking.

memory pages that store the LIFT code read-only. Thus, any attempt to modify the LIFT code causes a page fault.

To protect the tag space, LIFT uses a mechanism similar to prior work by Xu et al. [XBS06]. That is, it turns off the access permission of those pages that store the tag values of the tag space itself (note that the tag space is also a part of the virtual memory space, so there is also a tag bit for each byte of the tag space). Thus any instruction in the original program or some hijacked code accesses the tag space will result in some information flow tracking, which needs to access the corresponding tags and thereby triggers a protection fault.

## 4.3 LIFT Binary Optimizations

Last section described the baseline system of LIFT. Since it does not have any optimizations, similar to previous software-only runtime information flow tracking systems [NS05, CCC+05], it incurs large runtime overhead (up to 47 times as shown in our experiments). To minimize the over-

head associated with information flow tracking so that it is practical to use during production runs against security attacks, on top of the baseline system LIFT employs three binary optimizations including (1) Fast Path (FP) that eliminates unnecessary information flow tracking, (2) Merged Check (MC) that merges multiple tag checks into one, and (3) Fast Switch (FS) that reduces the overhead incurred for switching between instrumented code and the original program.

All the above optimizations do not sacrifice the capability of detecting security attacks because they are all conservative: never eliminate any necessary tag propagations. In addition, they are all performed at the binary level so they work for software and libraries whose source code is unavailable. Even though it is possible to implement the third optimization, FS, via static instrumentation, the FP and MC optimizations benefit from the trace linking (also referred as hot traces) mechanism (each trace combines multiple basic blocks dynamically) available only in dynamic instrumentation frameworks. The following three subsections describe the three optimizations, respectively.

### 4.3.1 Fast Path (FP) Optimization

The Fast-Path (FP) optimization is based on the observation that, for most applications, majority of tag propagations are zero-to-zero, i.e., from safe data sources to a safe destination. To validate the above hypothesis, I collect some statistics of a running Apache Web server. In the experiments, all data received from the network are tagged as one (unsafe). At runtime, LIFT collects statistics on the distribution of different types of tag propagations: (1) $S \rightarrow S$: both the sources and the destination are safe; (2) $S \rightarrow U$: a safe data overwrites an unsafe data in the destination; (3) $U \rightarrow S$: the instruction propagates an unsafe data to a memory location that stores safe data. and (4) $U \rightarrow U$: the instruction propagates an unsafe data to a memory location that stores unsafe data.

As shown on Figure 4.2, majority of tag propagation belongs to the first type: (1) $S \rightarrow S$. This is because, in most applications, only data received from network are tagged as unsafe initially, and most other data that do not have data dependency on these data will remain safe, for at least *many execution periods (even though it may not be always safe for the entire execution).* Therefore, any computation among these safe data corresponds to zero-to-zero tag propagation.

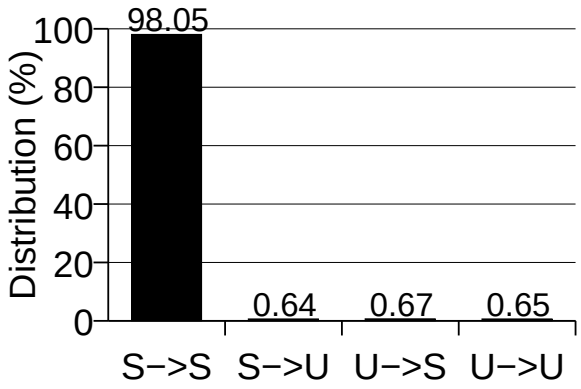



Figure 4.2: Distribution of four groups of tag propagation in Apache

The above observation provides a good dynamic optimization opportunity to eliminate unnecessary tag propagation. Specifically, before a code segment (either a basic block or a hot trace [BWWA06, LCM+05])), we can insert some checks to see if all its live-in and live-out registers or memory data are safe or not. If so, then there is no need to do any information flow tracking inside this code segment. Checking the safety of all live-ins at the very beginning of a code segment is very intuitive as they are the source operands. LIFT also needs to check the safety of all live-out locations at the very beginning of a code segment because they may currently store unsafe data, and may be overwritten by some safe data inside this code segment, in which case it is necessary to do information flow tracking inside this code segment. There is no need to check other data because they are either not used in this code segment, or they are dead at the very beginning or end of this code segment.

The Fast-Path (FP) optimization is based on the above idea. It inserts information checks before entering a code segment. If all live-ins and live-outs are safe, it runs the fast binary version, referred as the *check version*, without any information flow tracking. Otherwise, it runs the slow version, referred as the *track version*, which performs information flow tracking. By *dynamically* switching between fast and slow versions, LIFT can effectively avoid unnecessary information flow tracking for *dynamic* code segments (dynamic instances of code segments) that do not involve any unsafe data. Since it always performs tag checks first to decide whether to run the track version, it does not affect the capability of detecting security attacks.

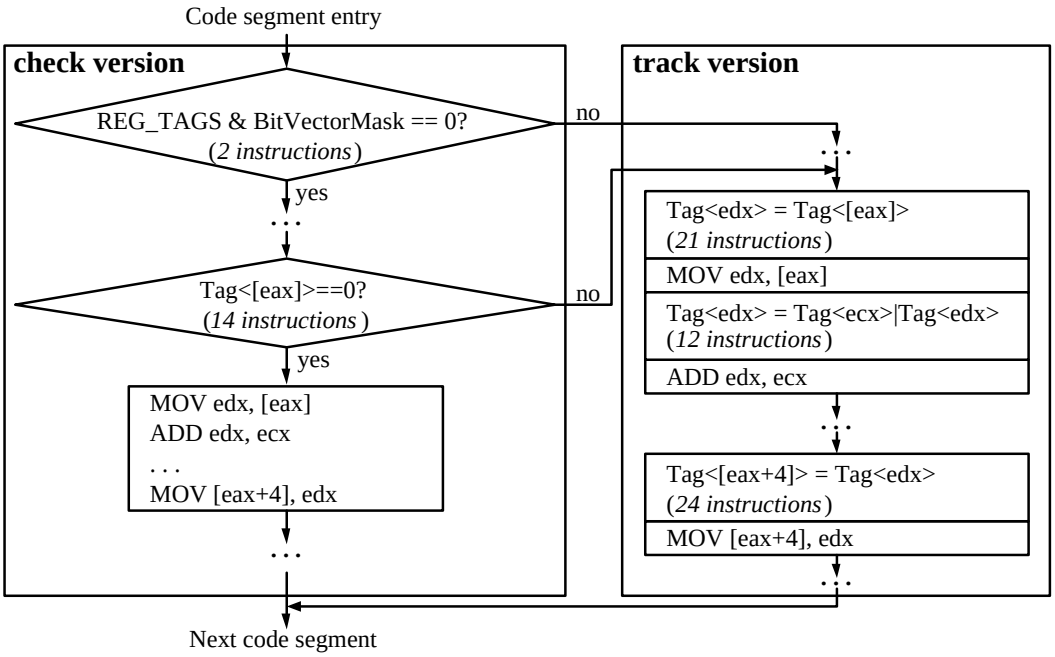



Figure 4.3: An example of the FP and MC optimizations. A code segment here can be a basic block or a hot trace, which is formed dynamically at runtime and can consist of multiple basic blocks.

LIFT can easily check the tags for all the registers used in a program region. Essentially, it associates with each code segment a bit vector, called $BitVectorMask$, which records the live-in and live-out registers whose tags need to be checked. As demonstrated in Figure 4.3, at the beginning of a code segment, a check is inserted by performing an AND operation on the $BitVectorMask$ and $REG\_TAGS$, which records the tags for all general data registers. If the result is zero, it follows the check version, otherwise jumps to the track version.

Unfortunately, to know the memory live-ins and live-outs at the beginning of a code segment are much harder because some addresses may not be known at the beginning of the code segment. Therefore, as demonstrated on Figure 4.3, to handle memory data tags, LIFT postpones the information check of a memory location until its address is known, usually right before this memory instruction. If its tag is zero, it continues the check version; otherwise, it jumps to the corresponding instruction in the track version.

The granularity of a code segment can be either a basic block, or a hot trace which is formed

dynamically at runtime and can consist of multiple basic blocks. At run time, if multiple basic blocks are frequently accessed one after another, the dynamic binary instrumentation engine will link them together by replacing indirect or conditional jump into a move and a direct jump. Then all these multiple basic blocks form a hot trace which is then stored in the trace cache. Obviously, it is better to perform the optimization at the trace granularity than at the basic granularity because the former performs only one check for registers in the very beginning of a trace and also provide opportunity for the next optimization, MC, to merge more memory tag checks into one.

The FP optimization can be further optimized by dynamic adaptation based on different behavior of code segments. For some code segments, the program may always execute the slow version. For example, those functions that receive and process network data always access "unsafe" data. In such cases, we can directly switch the program control flow to the slow version without wasting time to do the first few checks in the fast version.

### 4.3.2 Merged Check (MC) Optimization

Even after the FP optimization, many information checks are redundant or semi-redundant, which provides an opportunity for the second optimization: Merged Check(MC) optimization. MC reduces the number of information checks by combining multiple tag checks into one. Similar to the FP optimization, it is more beneficial to perform the MC optimization at the trace granularity.

To combine multiple checks into one, MC exploits both temporal locality and spatial locality of memory references commonly exhibited in many applications. Temporal locality says that a recently accessed data is likely to be accessed again in the near future, whereas spatial locality means that after an access to a location, memory locations that are nearby are also likely to be accessed in the near future. To exploit the temporal locality characteristic, if a trace has multiple memory references to the same location, MC combines the tag checks and performs it only once right before the first memory reference. Secondly, MC exploits the spatial locality of memory references and merge multiple tag checks of nearby memory locations into one check.

To perform the optimization, MC needs to find ahead of time what memory accesses are to

the same or nearby locations. It does this by performing memory reference analysis and then clustering the memory references into different groups. More specifically, MC first scans all the instructions in a trace and constructs a data dependency graph for each memory reference. The dependency graph for a memory reference consists of the version numbers of the registers and offsets for computing the address of this memory reference. It increments the version number of a register every time the register is defined by an instruction explicitly or implicitly. For example, the stack operations may implicitly modify the stack pointer register. From these dependency graphs, MC can easily cluster the nearby/same memory references into a group. For example, if multiple references depend on the same version of the same register and the same offset, they are to the same memory location. And if their offsets differ by a small number, they are to nearby memory locations. At the end, MC inserts one tag check before the first instruction of each group.

### 4.3.3 Fast Switch (FS) Optimization

In most general instrumentation frameworks such as PIN [LCM+05] and StarDBT [BWWA06], when the program execution switches between the original binary code and the instrumented code, it requires saving and restoring the context, including the condition register and the runtime stack registers (switching to a separate stack). The reason for saving the condition register before switching to the instrumented code is straightforward, the instrumented code may change the value of the condition register. The reason for using a separate stack for the instrumented code, i.e. the information flow tracking/checking code, is for avoiding modifying the stack in case of register overflow when executing the instrumented code. More explanation about this context switch requirement and process can be found in previous work [BWWA06, LCM+05].

This context switch, even though much smaller than OS-level context switches, can still introduce large runtime overhead, especially the instrumentation is inserted at many locations as in our case for information flow tracking. LIFT minimizes the above context switch overhead in the original StarDBT binary instrumentation framework by cleverly selecting cheaper instructions and performing liveness analysis.

First, the FS optimization of LIFT reduces the overhead associated with each context switch. Similar to other binary instrumentation tool, the original StarDBT saves/restores the context of the original code to/from the stack using simple but expensive $pushfq/popfq$ instructions in the x86 architecture. To make each context switch cheaper, LIFT uses two cheaper instructions $lahf/sahf$ to save/restore the condition register to other free registers. By eliminating $pushfq/popfq$, it also avoids the need of a separate stack for executing the information flow tracking/checking.

Second, with the FS optimization, LIFT performs condition register liveness analysis to eliminate those unnecessary condition register save and restore operations. In the x86 architecture, an $eflags$ register saves the program conditional flags. LIFT's liveness analysis tracks the define and use of eflags bits for each instruction within a program region of the original code. In many cases, the eflag register value is dead at the beginning of many program regions (e.g. instruction traces). Therefore, it is unnecessary to save it before switching to the instrumented code, i.e. the information tracking or checking code.

## 4.4 Evaluation Methodology

**Test Platform** The experiments are conducted on real machines. The evaluated applications runs on an EM64T machine with two 64-bit Xeon processors of 3.0GHz, 512KB L2 cache, and 1GB memory, running the Windows XP 64-bit version as the OS. For the network applications, we also use a second machine to act as the other party of the evaluated application. This machine has two Xeon processors of 2.2 GHz, 512KB L2 cache, and 512MB memory, runs the Linux 2.6.9 kernel and is connected to the EM64T machine via 100Mbps Ethernet network. I implement LIFT on StarDBT [BWWA06], a dynamic binary translator developed by Intel.

**Applications** We evaluate the functionality and performance of LIFT with a variety of applications, including three real-world network applications (two servers and one client) and two benchmark suits. The network applications include Apache Web server [Apa06], Savant Web

| Applications | Version | Exploits | App Description |
|---|---|---|---|
| Apache | 1.3.24 | overwrite the function pointer | a web server |
| Savant | 3.1 | overwrite the return address | a web server |
| Putty | 0.53 | overwrite the return address | a telnet program |
| Attack Benchmarks | 2003 | 18 different types of exploits | a buffer overflow testbed |

Table 4.2: Applications and security exploits

server [sav06] and Putty [Tat06]. The first benchmark suite consists of eighteen different attack benchmarks developed by John Wilander [WK03] and covers a variety of different security exploits. We ported the attack benchmarks from Linux version to Windows version.

To evaluate LIFT's capability in detecting general types of security attacks, we use three network applications as well as the eighteen attack benchmarks, as listed on Table 4.2. To play the real-world attacks for the three network applications, we leverage the Metasploit [met06] framework to send the malicious inputs. The experiments cover a variety of different exploits. For example, the exploit in Savant Web server overwrites the return address in the stack. The attack benchmarks cover eighteen types of exploiting methods, including different overwrite techniques (direct or indirect), different buffer locations (stack or heap/BSS/data), and different attack targets (return address, base pointer, function pointer, or longjmp buffers).

For real-world network applications, we use Windows Layered Service Provider [HOB99] technique to intercept network data and tag received data as "unsafe". This tagger works in the network layer and requires no source code of target programs. Since the attack benchmarks simulate network input, we have to modify the testbed to tag the simulated network input data as "unsafe".

To evaluate LIFT's overhead and the effects of our optimizations on latency and throughput, we use seven SPEC INT2000 benchmark and the Apache Web server. For Apache, we label all data received from network as "unsafe" and measure the throughput. For SPEC benchmarks, we measure the performance for two configurations of initial tagging scheme: one is tagging all the input data from disk files as "unsafe" for simulate network data; the other is tagging all the input data from disk files as "safe" to measure the performance upper bounds.

| Exploits Targets (Exploits #) | Detected #/Exploits # | | | | |
|---|---|---|---|---|---|
| | StackGuard | Stack Shield | ProPolice | LibSafe and Libverify | LIFT |
| Return Address (3) | 3/3 | 3/3 | 2/3 | 1/3 | 3/3 |
| Base Pointer (3) | 2/3 | 3/3 | 2/3 | 1/3 | 3/3 |
| Function Pointer (6) | 0/6 | 0/6 | 3/6 | 1/6 | 6/6 |
| Longjmp buffer (6) | 0/6 | 0/6 | 3/6 | 1/6 | 6/6 |
| Total (18) | 5/18 | 6/18 | 10/18 | 4/18 | 18/18 |

Table 4.3: Results of LIFT for attack benchmarks

## 4.5 Experimental Results

### 4.5.1 Security Attack Detection

Table 4.3 shows the effectiveness of LIFT in detecting general security attacks. We compare LIFT's results with those reported by prior work [WK03] that evaluated five existing tools, including StackGuard [CPM+98], Stack Shield [sta06], ProPolice [Eto06], and LibSafe+LibVerify [BTS99, BST00], using the same eighteen attack benchmarks. We classify the eighteen types of attacks into four groups based on their exploiting targets, including return address, base pointer, function pointer, and longjmp buffer. Those targets can be either in the stack or heap/BBS/data regions.

Overall, LIFT detects all the eighteen exploits of various types because it is oblivious to the specific exploit method such as smashing a return address, overwriting a function pointer, etc. All these exploit methods need to switch the program control to some "unsafe" data in order to hijack the program, so they are all detected by LIFT. In contrast, the other five tools shown in Table 4.3 can only detect some of the exploits since they are designed for certain types of exploits and cannot deal with other unknown exploits. For example, StackGuard and Stack Shield can only detect those attacks that try to smash a return address and a base pointer.

The evaluation with three real-world network applications, including two popular Web servers (Apache and Savant) and one network client (Putty), shows that LIFT can also detect various types of attacks in real-world scenario. For example, the vulnerabilities in Savant Web server and Putty are exploited to overwrite the return address and switch the program control to some "unsafe" code.

| Configurations | Throughput | | Response Time | |
|---|---|---|---|---|
| | Results (MBps) | Overhead (%) | Results (millisecond) | Overhead (%) |
| Native | 8.06 | 0 | 1.1 | 0 |
| StarDBT | 7.79 | 3.4 | 1.5 | 36.4 |
| LIFT-basic | 6.40 | 20.6 | 5.1 | 363.6 |
| LIFT-FS | 6.97 | 13.6 | 3.5 | 220.0 |
| LIFT-FS-FP | 7.49 | 7.1 | 2.3 | 109.1 |
| LIFT | 7.56 | 6.2 | 2.1 | 90.9 |

Table 4.4: The throughput and response time of Apache running on native machine and with different optimizations techniques applied. "Native" means that the Apache runs directly on the machine, "StarDBT" refers to our base line binary translation framework without any LIFT-related instrumentation. "LIFT-basic" is the basic LIFT system without any optimizations. "LIFT-FS" is LIFT-basic with the Fast-Switch optimization. "LIFT-FS-FP" is LIFT-basic with Fast-Switch and Fast-Path optimizations, and "LIFT" is LIFT-basic with all three optimizations. The requested file sizes uniformly distributed among 4KB, 8KB, 16KB, to 512KB.

LIFT successfully reports these two attacks. For Apache, a long request overwrites the whole stack and triggers an exception when attempting to write beyond the stack boundary. The default signal handler in a system library fetches a function pointer from the corrupted stack and switch the program control via that "unsafe" function pointer. In the experiments, we observed LIFT marks all the data in the corrupted stack as "unsafe" and theoretically it can catch this attack once the program control is switched via the "unsafe" function pointer. However, LIFT does not report this attack since the current version of StarDBT does not provide accurate exception handling. We are improving StarDBT on this issue.

LIFT raises no false alarms in all the experiments. We run LIFT normally with network applications such as Apache Web server and Putty and other small utilities such as Notepad without any false alarms reported. In addition, we tag all the input data from disk files for the tested SPEC-INT programs as "unsafe", LIFT still runs through all the tests without raising any false alarms.

### 4.5.2 Performance Results

#### Overhead with Apache

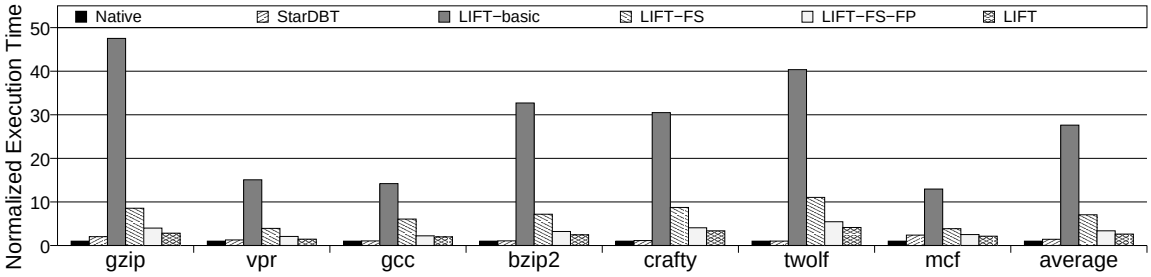
Table 4.4 shows that LIFT incurs low runtime overhead for Apache. With all three optimizations, LIFT incurs only 6.2% for the throughput of Apache, close to 3.4% incurred by StarDBT. This is because LIFT aggressively applies dynamic optimization to eliminate unnecessary information flow tracking and provide a fast switch between the instrumented code and the original code. For example, the Fast Switch (FS) optimization reduce the overhead for the throughput from 20.6% to 13.6% and the overhead for the response time from 363.6% to 220%. The Fast Path (FP) optimization further improves the performance, bringing down the overhead for the throughput to 7.1% and the overhead for the response time to 109.1%.

The overhead of LIFT comes from several sources. The first is the StarDBT binary translation framework which incurs 3.4% overhead. The second source comes from the dynamic translating, instrumenting, optimizing and maintaining the binary code. The third source, the most significant one, is the overhead for executing the instrumented code to perform tag checks, tag propagation, and attack detection.

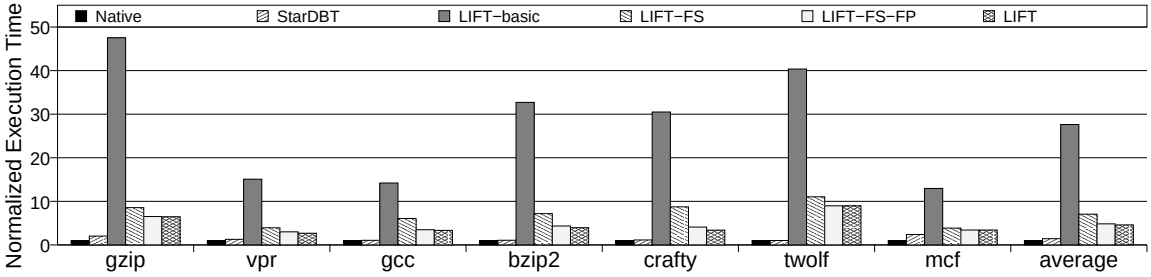
#### Overhead with SPEC INT Benchmarks

Figure 4.4 shows that LIFT incurs low runtime overhead for the seven SPEC programs. Benefited from the three optimizations, LIFT incurs only 1.7-7.9 times overhead and an average of 3.6 times overhead when all the input data are tagged “unsafe”, much smaller than the large overhead (40 times) slowdowns reported for a previous binary instrumentation-based information flow tracking tool [NS05]. This is because LIFT aggressively applies dynamic optimization to eliminate unnecessary information flow tracking and provide a fast switch between instrumented code and the original code.

Figure 4.4 also shows that the three optimizations effectively reduce the overhead incurred in the basic LIFT system. For example, without any optimization, LIFT-basic slows down the




**(a) all input data are tagged safe**




**(b) all input data are tagged unsafe**

Figure 4.4: Overall Results. Comparison of normalized execution time. "Native", "StarDBT", "LIFT-basic", "LIFT-FS", "LIFT-FS-FP", and "LIFT" have the same meaning as in Table 4.4. Initially, the input data from disk files are tagged safe or unsafe in the two figures respectively.

program execution by 12.0-46.5 times and on average 26.6 times, which are effectively reduced by the three dynamic optimizations to an average of 3.6 times overhead, a factor of 7.4 times reduction in overhead!

With input data initially tagged as "safe" or "unsafe", LIFT-basic shows no difference in terms of runtime overhead since the tag is propagated regardless whether it is "safe" or not. In contrast, LIFT, with all optimizations, does incur different overheads. For example, with all input data tagged as "safe" for vpr, LIFT incurs only 0.6 times overhead, and with all input data tagged as "unsafe" for vpr, it incurs 1.7 times overhead. This is because, if all input data are tagged "safe", LIFT always run the check version, which is much faster than the track version. Note here, that even with all input data tagged "unsafe", LIFT does not necessarily always run the track version since there still exist many computation that do not involve any "unsafe" data (because usually information flow tracking systems do not track control dependencies [CCC+05, NS05, VBC+04, SLZD04]).

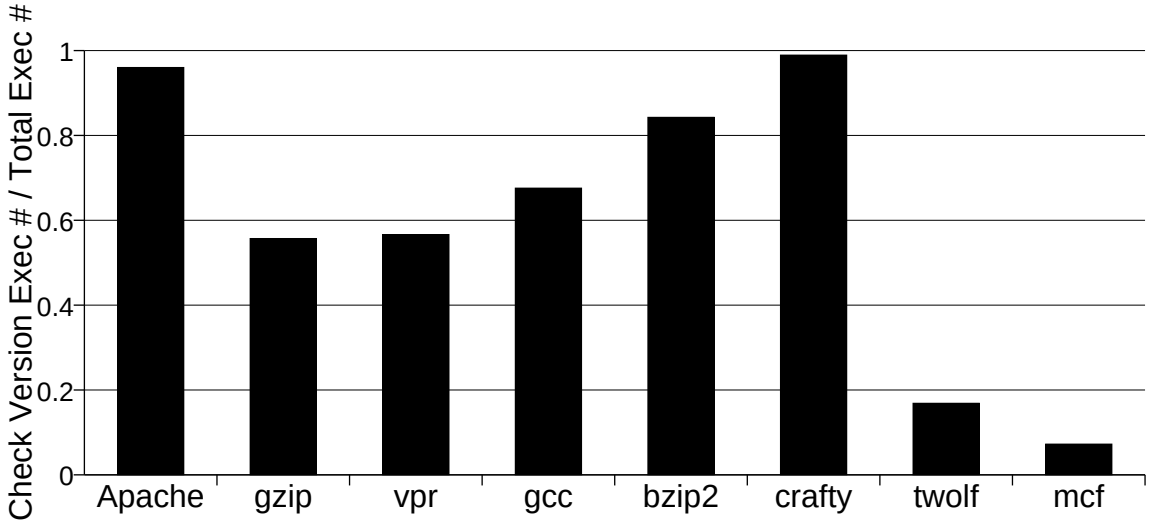



Figure 4.5: The execution number of the check version and the track version for basic blocks in Apache and SPEC. For SPEC, all the input data are tagged as "unsafe". The total execution number is the sum of the execution number of both the check version and the track version.

**Effects of Optimizations**

Figure 4.4 shows that the three optimizations can effectively reduce the runtime overhead caused by LIFT-basic. Now let us examine the effect of each individual optimization. First, we apply the FS optimization to LIFT-basic since LIFT-basic has very frequent and heavy context switches. With all input data tagged as unsafe, the FS optimization can reduce the overhead significantly by a factor of 4.4 times. This is because it reduces both the cost of each context switching by using cheaper instructions and the number of context switches by using eflag liveness analysis.

The FP optimization reduces the overhead incurred by LIFT-FS for all applications to different extent. For example, with all input data tagged as "unsafe" initially, OPT-FP further reduces the overhead of LIFT-FS for crafty from 7.7 times to 3.1 times, while reducing the overhead of LIFT-FS for mcf from 2.9 times to 2.4 times. This is because the amount of overhead reduction depends on the percentage of check versions are executed for each application. As shown in Figure 4.5, with all input data are tagged as "unsafe", a significant amount (98.9%) of the execution for crafty

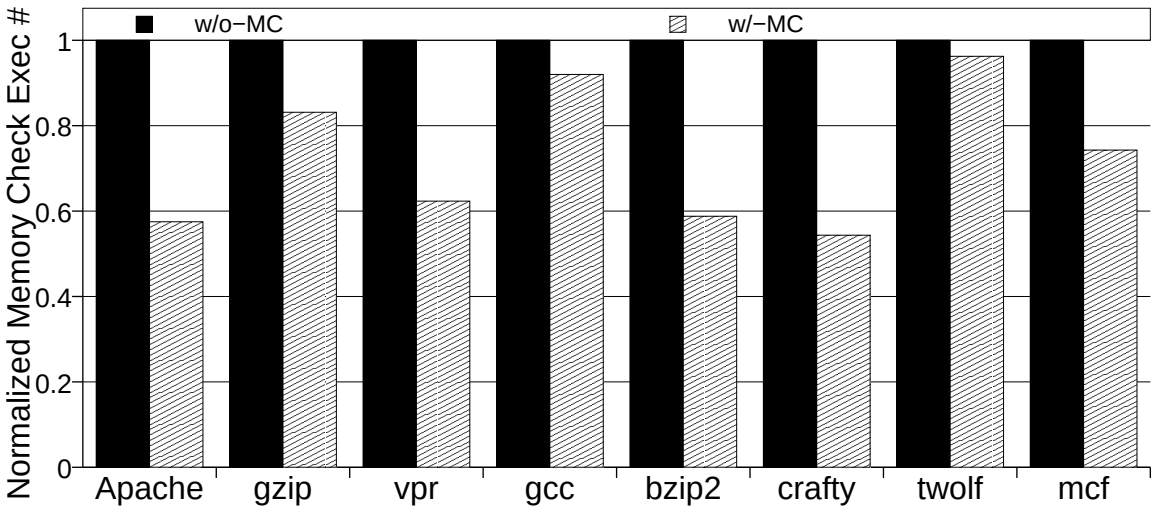



Figure 4.6: The execution number of memory checks for SPEC. All the input data are tagged as "unsafe". "w/o-MC" means run LIFT-FS-FP without the MC optimization applied. "w/-MC" refers to LIFT with all three optimizations including MC.

is in the check version, which results in the large reduction of the overhead incurred by LIFT-FS, while mcf only has 7.2% of the execution in the check version.

MC further reduces the overhead of LIFT after the first two optimizations, FS and FP, for all cases with different percentage. For example, the overhead of Apache's throughput decreased from 7.1% to 6.2% after applying MC. For SPEC applications the overhead reduction varies. For example, MC reduce the overhead for crafty from 3.1 times to 2.4 times, while it has no visible effects on the overhead for twolf (reducing from 7.97 times to 7.96 times). This is because the overhead reduction depends on how many executed memory checks are reduced and how many percentage the program execute the fast version since only the fast version contains the memory checks. Figure 4.6 shows the normalized executed memory checks number with and without applying MC with all the input data tagged as "unsafe". We can see that MC reduced 46% of executed memory checks for crafty, while it only reduced 4% of executed memory checks for twolf.

## 4.6 Summary

In summary, LIFT is a low-overhead, cheap (no hardware extension), comprehensive (works with libraries), and practical information flow tracking system for detecting general security attacks. It is suitable for building a second-level defense during production runs. It minimizes runtime overhead by exploiting dynamic binary instrumentation and optimization including the Fast-Path, Merged-Check and Fast-Switch optimizations.

The real-system experiments with three real-world network applications, including two Web servers and one client application, and *eighteen* attack benchmarks show that LIFT can effectively detect all tested 21 security attacks of various types, much more than five of the previous tools that can only detect at most ten attacks as shown in prior study [WK03]. More importantly, compared to other software-only binary-based information flow tracking system that slows down program execution by more than 40 times [NS05], LIFT incurs significantly less overhead, only 6.2% for server applications, and 3.6 times on average for seven SPEC INT2000 applications. The three optimizations also effectively reduce the overhead by a factor of 5-12 times.

# Chapter 5

# Third-Level Defense: Surviving Software Failures

## 5.1 Overview

### 5.1.1 Motivation

After having detected software bugs in the first level of defense and exploitation of software bugs in the second level of defense as demonstrated in Chapter 3 and Chapter 4 respectively, the program can be simply shut down to prevent further damage to the system. However, many applications, especially critical ones such as process control or on-line transaction monitoring, require high availability [Gra86]. This chapter proposes a *safe* technique, called *Rx*, to build the third level of defense via quickly recovering from many types of software failures caused by common software bugs, both deterministic and non-deterministic.

As reviewed in Chapter 2, much research has been conducted in surviving software failures. This dissertation classifies them into four categories. The first category encompasses various flavors of rebooting (restarting) techniques, including whole program restart [Gra86, SC91], micro-rebooting of partial system components [CCF+02, CKF+04], and software rejuvenation [HKKF95, GPTT97, BS98]. Since many of these techniques were originally designed to handle *hardware* failures, most of them are ill-suited for surviving software failures. For example, they cannot deal with deterministic software bugs, a major cause of software failures [CC00], because these bugs will still occur even after rebooting. Another major limitation of these methods is service unavailability while restarting, which can take up to several seconds [VDA+98]. For servers that buffer significant amount of state in main memory (e.g. data buffer caches), it requires a long period to warm up

to full service capacity [BBG+89, VDB+98]. Micro-rebooting [CKF+04] addresses this problem to some extent by only rebooting the failed components. However, it requires legacy software to be reconstructed in a loosely-coupled fashion.

The second category includes general checkpointing and recovery. The most straightforward method in this category is to checkpoint, rollback upon failures, and then re-execute either on the same machine [EAWJ02, RLT78] or on a different machine designated as the "backup server" (either active or passive) [Gra86, Bar81, BBG+89, BS96, VDB+98, ACZ00, ZCL99]. Similar to the whole program restart approach, these techniques were also proposed to deal with hardware failures, and therefore suffer from the same limitations in addressing software failures. In particular, they also cannot deal with failures caused by deterministic bugs. Progressive retry [WHF93] is an interesting improvement over these approaches. It reorders messages to increase the degree of non-determinism. While this work proposes a promising direction, it limits the technique to message reordering. As a result, it cannot handle bugs unrelated to message order. For example, if a server receives a malicious request that exploits a buffer overflow bug, simply reordering messages will not solve the problem. The most aggressive approaches in the checkpointing/recovery category include recovery blocks [Ran75] and n-version programming [AC77, Avi85, RCL01], both of which rely on different implementation versions upon failures. These approaches may be able to survive deterministic bugs under the assumption that different versions fail independently. But they are too expensive to be adopted by software companies because they double the software development costs and efforts.

The third category comprises application-specific recovery mechanisms, such as the multi-process model (MPM), exception handling, etc. Some multi-processed applications, such as the old version of the Apache HTTP Server and the CVS server, spawn a new process for each client connection and therefore can simply kill a failed process and start a new one to handle a failed request. While simple and capable of surviving certain software failures, this technique has several limitations. First, if the bug is deterministic, the new process will most likely fail again at the same place given the same request (e.g. a malicious request). Second, if a shared data structure is

corrupted, simply killing the failed process and restarting a new one will not restore the shared data to a consistent state, therefore potentially causing subsequent failures in other processes. Other application-specific recovery mechanisms require software to be failure-aware, which adversely affects programming difficulty and code readability.

The fourth category includes several recent non-conventional proposals such as failure-oblivious computing [RCD+04] and the reactive immune system [SLBK05]. Failure-oblivious computing proposes to deal with buffer overflows by providing *artificial* values for out-of-bound reads, while the reactive immune system returns a *speculative* error code for functions that suffer software failures (e.g. crashes). While these approaches are fascinating and may work for certain types of applications or certain types of bugs, they are *unsafe* to use for correctness-critical applications (e.g. on-line banking systems) because they "speculate" on programmers' intentions, which can lead to program misbehavior. The problem becomes even more severe and harder to detect if the speculative "fix" introduces a silent error that does not manifest itself immediately. In addition, such problems, if they occur, are very hard for programmers to diagnose since the application's execution has been forcefully perturbed by those speculative "fixes".

Besides the above individual limitations, existing work provides insufficient feedback to developers for debugging. For example, the information provided to developers may include only a core dump, several recent checkpoints, and an event log for deterministic replay of a few seconds of recent execution. To save programmers' debugging effort, it is desirable if the run-time system can provide information regarding the bug type, under what conditions the bug is triggered, and how it can be avoided. Such diagnostic information can guide programmers during their debugging process and thereby enhance their efficiency.

### 5.1.2 Highlights of Rx

This dissertation proposes a *safe* (not speculatively "fixing" the bug) technique, called *Rx*, to build the third level of defense via quickly recovering from many types of software failures caused by common software defects, both deterministic and non-deterministic. It requires few to no changes

to applications' source code, and provides diagnostic information for postmortem bug analysis. The idea is to rollback the program to a recent checkpoint when a bug is detected, *dynamically change the execution environment based on the failure symptoms*, and then re-execute the buggy code region in the new environment. If the re-execution successfully pass through the problematic period, the new environmental changes are disabled to avoid imposing time and space overheads.

The idea of Rx is inspired from real life. When a person suffers from an allergy, the most common treatment is to remove the allergens from her/his *living environment*. For example, if patients are allergic to milk, they should remove diary products from the diet. If patients are allergic to pollen, they may install air filters to remove pollen from the air. Additionally, when removing a candidate allergen from the environment successfully treats the symptoms, it allows diagnosis of the cause of the symptoms. Obviously, such treatment cannot and also should not start before patients shows allergic symptoms since changing living environment requires special effort and may also be unhealthy.

In software, many bugs resemble allergies. That is, their manifestation can be avoided by *changing the execution environment*. According to a previous study by Chandra and Chen [CC00], around 56% of faults in Apache HTTP server depend on execution environment [1]. Therefore, by removing the "allergen" from the execution environment, it is possible to avoid such bugs. For example, a memory corruption bug may disappear if the memory allocator delays the recycling of recently freed buffers or allocates buffers non-consecutively in isolated locations. A buffer overrun may not manifest itself if the memory allocator pads the ends of every buffer with extra space. Uninitialized reads may be avoided if every newly allocated buffer is all filled with zeros. Data races can be avoided by changing timing related events such as thread-scheduling, asynchronous events, etc. Bugs that are exploited by malicious users can be avoided by dropping such requests during program re-execution. Even though dropping requests may make a few users (hopefully the malicious ones) unhappy, they do not introduce incorrect behavior to program execution as

[1] Note that the definition of execution environment in this dissertation is different from theirs. Here, the standard library calls, such as $malloc$, and system calls are also part of execution environment.

the failure-oblivious approaches do. Furthermore, given a spectrum of possible environmental changes, the least intrusive changes can be tried first, reserving the more extreme one as a last resort for when all other changes have failed. Finally, the specific environmental change which cures the problem gives diagnostic information as to what the bug is.

Similar to an allergy, it is difficult and expensive to apply these execution environmental changes from the very beginning of the program execution because we do not know what bugs might occur later. For example, zero-filling newly allocated buffers imposes time overhead. Therefore, Rx should lazily apply environmental changes only when needed.

We have implemented Rx with Linux and evaluated it with four server applications that contain four real bugs (bugs introduced by the original programmers) and two injected bugs (bugs injected by us) of various types including buffer overflow, double free, stack overflow, data race, uninitialized read and dangling pointer bugs. Compared with previous solutions, Rx has the following unique advantages:

- *Comprehensive:* Rx can survive many common software defects. Besides non-deterministic bugs, Rx can also survive deterministic bugs. The experiments show that Rx can successfully survive the six bugs listed above. In contrast, the two tested alternatives, a whole program restart approach and a simple rollback and re-execution without environmental changes, cannot recover the three servers (Squid, Apache, and CVS) that contain deterministic bugs, and have only a 40% recovery rate for the server (MySQL) that contains a non-deterministic concurrency bug. Such results indicate that applying environmental changes during re-execution is the key reason for Rx's successful recovery of all tested cases.

- *Safe:* Rx does not speculatively "fix" bugs at run time. Instead, it prevents bugs from manifesting themselves by changing only the program's execution environment. Therefore, it does not introduce uncertainty or misbehavior into a program's execution, which is usually very difficult for programmers to diagnose.

- *Noninvasive:* Rx requires few to no modifications to applications' source code. Therefore, it

can be easily applied to legacy software. In our experiments, Rx successfully avoids software defects in the four tested server applications without modifying any of them.

- *Efficient:* Because Rx requires no rebooting or warm-up, it significantly reduces system down time and provides reasonably good performance during recovery. In the experiments, Rx recovers from software failure within 0.017-0.16 seconds, 21-53 times faster than the whole program restart approach for all but one case (CVS). Such efficiency enables servers to provide non-stop services despite software failures caused by common software defects. Additionally, Rx is quite efficient. The technology imposes little overhead on server throughput and average response time and also has small space overhead.

- *Informative:* Rx does not hide software bugs. Instead, bugs are still exposed. Furthermore, besides the usual bug report package (including core dumps, checkpoints and event logs), Rx provides programmers with additional diagnostic information for postmortem analysis, including what conditions triggered the bug and which environmental changes can or cannot avoid the bug. Based on such information, programmers can more efficiently find the root cause of the bug. For example, if Rx successfully avoids a bug by padding newly allocated buffers, the bug is likely to be a buffer overflow. Similarly, if Rx avoids a bug by delaying the recycling of freed buffers, the bug is likely to be caused by double free or dangling pointers.

The rest of this chapter is organized as follows. Section 5.2 presents the main idea of Rx. Followed by Rx design in Section 5.3. Section 5.4 discusses issues in design and implementation. Section 5.5 and Section 5.6 presents the evaluation methodology and results respectively. Section 5.7 summarize this chapter.

## 5.2 Main Idea of Rx

The main idea of Rx is to, upon a software failure, rollback the program to a recent checkpoint and re-execute it in a new environment that has been modified based on the failure symptoms.

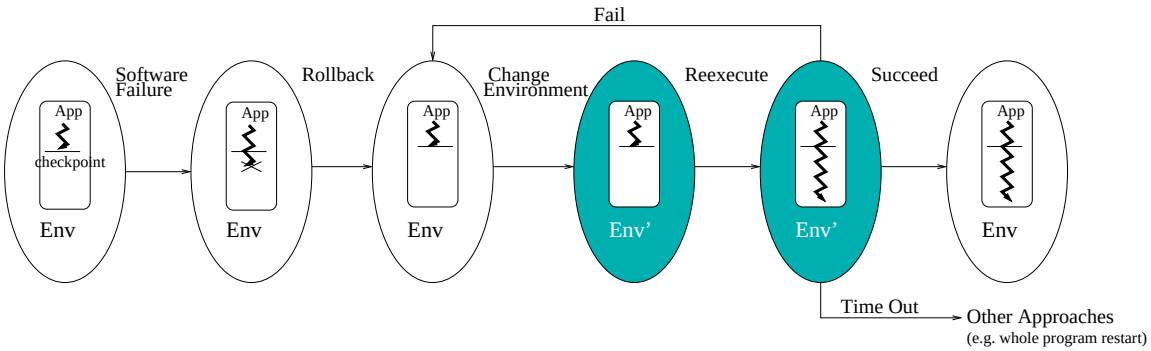



Figure 5.1: Rx: The main idea

If the bug's "allergen" is removed from the new environment, the bug will not occur during re-execution, and thus the program will survive this software failure. After the re-execution safely passes through the problematic code region, the environmental changes are disabled to reduce time and space overhead imposed by the environmental changes.

Figure 5.1 shows the process by which Rx survives software failures. Rx periodically takes light-weight checkpoints that are specially designed to survive software failures instead of hardware failures or OS crashes (See Section 5.3.2). When a bug is detected, either by an exception or by integrated dynamic software bug detection tools called as the Rx sensors, the program is rolled back to a recent checkpoint. Rx then analyzes the occurring failure based on the failure symptoms and "experiences" accumulated from previous failures, and determines how to apply environmental changes to avoid this failure. Finally, the program re-executes from the checkpoint in the modified environment. This process will repeat by re-executing from different checkpoints and applying different environmental changes until either the failure does not recur or Rx times out, resorting to alternate solutions, such as whole-program rebooting [Gra86, SC91]. If the failure does not occur during re-execution, the environmental changes are disabled to avoid the overhead associated with these changes.

In Rx idea, the execution environment can include almost everything that is external to the target application and can affect the execution of the target application. At the lowest level, it includes the hardware such as processor architectures, devices, etc. At the middle level, it includes the OS kernel such as scheduling, virtual memory management, device drivers, file systems, network pro-

| Category | Environmental Changes | Potentially-Avoided Bugs | Deterministic? |
|---|---|---|---|
| Memory Management | delayed recycling of freed buffer | double free, dangling pointer | YES |
| | padding allocated memory blocks | dynamic buffer overflow | YES |
| | allocating memory in an alternate location | memory corruption | YES |
| | zero-filling newly allocated memory buffers | uninitialized read | YES |
| Asynchronous | scheduling | data race | NO |
| | signal delivery | data race | NO |
| | message reordering | data race | NO |
| User-Related | dropping user requests | bugs related to the dropped request | Depends |

Table 5.1: Possible environmental changes and their potentially-avoided bugs

tocols, etc. At the highest level, it includes standard libraries, third-party libraries, etc. Such definition of the execution environment is much broader than the one used in previous work [CC00].

Obviously, the execution environment cannot be arbitrarily modified for re-execution. A useful re-execution environmental change should satisfy two properties. First, it should be *correctness-preserving*, i.e., every step (e.g., instruction, library call and system call) of the program is executed according to the APIs. For example, in the $malloc()$ library call, we have the flexibility to decide where buffers should be allocated, but we cannot allocate a smaller buffer than requested. Second, a useful environmental change should be able to potentially avoid some software bugs. For example, padding every allocated buffer can prevent some buffer overflow bugs from manifesting during re-execution.

Table 5.1 lists some environmental changes and the types of bugs that can be potentially avoided by them. Examples of useful execution environmental changes include, but are not limited to, the following categories:

**(1)Memory management based**: Many software bugs are memory related, such as buffer overflow, dangling pointers, etc. These bugs may not manifest themselves if memory management is performed slightly differently. For example, each buffer allocated during re-execution can have padding added to both ends to prevent some buffer overflows. Delaying the recycling of freed buffers can reduce the probability for a dangling pointer to cause memory corruption. In addition, buffers allocated during re-execution can be placed in isolated locations far away from existing memory buffers to avoid some memory corruption. Furthermore, zero-filling new buffers can avoid some uninitialized read bugs. *Since none of the above changes violate memory allocation*

*or deallocation interface specifications, they are safe to apply*. Also note that these environmental changes affect only those memory allocations/deallocations made during re-execution.

**(2)Timing based**: Most non-deterministic software bugs, such as data races, are related to the timing of asynchronous events. These bugs will likely disappear under different timing conditions. Therefore, Rx can forcefully change the timing of these events to avoid these bugs during re-execution. For example, increasing the length of a scheduling time slot can avoid context switches during buggy critical sections. This is very useful for those concurrency bugs that have high probability of occurrences. For example, the data race bug of the MySQL server in the experiments has a 40% occurrence rate on a uniprocessor machine.

**(3)User request based**: Since it is infeasible to test every possible user request before releasing software, many bugs occur due to unexpected user requests. For example, malicious users issue malformed requests to exploit buffer overflow bugs during stack smashing attacks [CPM+98]. These bugs can be avoided by dropping some users' requests during re-execution. Of course, since the user may not be malicious, this method should be used as a last resort after all other environmental changes fail.

If the failure disappears during a re-execution attempt, the failure symptoms and the effects of the environmental changes applied are recorded. This speeds up the process of dealing with future failures that have similar symptoms and code locations. Additionally, Rx provides all such diagnostic information to programmers together with core dumps and other basic postmortem bug analysis information. For example, if Rx reports that buffer padding does not avoid the occurring bug but zero-filling newly allocated buffers does, the programmer knows that the software failure is more likely to be caused by an uninitialized read instead of a buffer overflow.

After a re-execution attempt successfully passes the problematic program region for a threshold amount of time, all environmental changes applied during the successful re-execution are disabled to reduce space and time overheads. These changes are no longer necessary since the program has safely passed the "allergic seasons".

If the failure still occurs during a re-execution attempt, Rx will rollback and re-execute the program again, either with a different environmental change or from an older checkpoint. For example, if one change (e.g. padding buffers) cannot avoid the bug during the re-execution, Rx will rollback the program again and try another change (e.g. zero-filling new buffers) during the next re-execution. If none of the environmental changes work, Rx will rollback further and repeat the same process. If the failure still remains after a threshold number of iterations of rollback/re-execute, Rx will resort to previous solutions, such as whole program restart [Gra86, SC91], or micro-rebooting [CCF+02, CKF+04] if supported by the application.

Upon a failure, Rx follows several rules to determine the order in which environmental changes should be applied during the recovery process. First, if a similar failure has been successfully avoided by Rx before, the environmental change that worked previously will be tried first. If this does not work, or if no information from previous failures exists, environmental changes with small overheads (e.g. padding buffers) are tried before those with large overheads (e.g. zero-filling new buffers). Changes with negative side effects (e.g. dropping requests) are tried last. Changes that do not conflict, such as padding buffers and changing event timing, can be applied simultaneously.

Although the situation never arose during the experiments, there is still the rare possibility that a bug still occurs during re-execution but is not detected in time by Rx's sensors. In this case, Rx will claim a recovery success while it is not. Addressing this problem requires using more rigorous on-the-fly software defect checkers as sensors. This is currently a hot research area that has attracted much attention. In addition, it is also important to note that, *unlike in failure oblivious computing, this problem is caused by the application's bug instead of Rx's environmental changes*. The environmental changes just make the bug manifest itself in a different way. Furthermore, since Rx logs its every action including what environmental changes are applied and what the results are, programmers can use this information (i.e. some environmental changes make the software crash much later) to analyze the occurring bug.

## 5.3 Rx Design

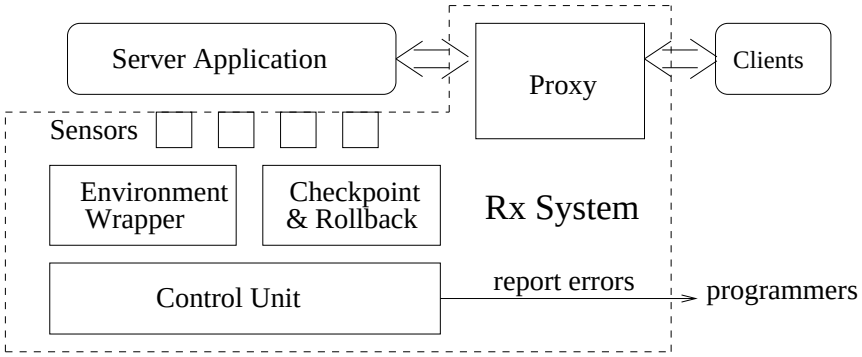



Figure 5.2: Rx architecture

Rx is composed of a set of user-level and kernel-level components that monitor and control the execution environment. The five primary components are seen in Figure 5.2: (1) sensors for detecting and identifying software failures or software defects at run time, (2) a Checkpoint-and-Rollback (CR) component for taking checkpoints of the target server application and rolling back the application to a previous checkpoint upon failure, (3) environment wrappers for changing execution environments during re-execution, (4) a proxy for making server recovery process transparent to clients, and (5) a control unit for maintaining checkpoints during normal execution, and devising a recovery strategy once software failures are reported by sensors.

### 5.3.1 Sensors

Sensors detect software failures by dynamically monitoring applications' execution. There are two types of sensors. The first type detects software errors such as assertion failures, access violations, divide-by-zero exceptions, etc. This type of sensor can be implemented by taking over OS-raised exceptions. The second type of sensor detects software bugs in the first level of defense or exploitation of software bugs in the second level of defense as discussed in Chapter 3 and Chapter 4 before they cause the program to crash. This type of sensors leverage existing low-overhead dynamic bug detection or bug exploits detection tools in the first or second level of defense, such as CCured [CHM+03b], StackGuard [CPM+98], SafeMem [QLZ05], and LIFT, to name a few. In

the Rx prototype, I have only implemented the first type of sensors. However, I plan to integrate second type of sensors into Rx.

Sensors notify the control unit upon software failures with information to help identify the occurring bug for recovery and also for postmortem bug diagnosis. Such information includes the type of exception (Segmentation fault, Floating Point Exception, Bus Error, Abort, etc.), the address of the offending instruction, stack signature, etc.

### 5.3.2 Checkpoint and Rollback

#### Mechanism

The CR (Checkpoint-and-Rollback) component takes checkpoints of the target server application, and automatically and transparently rolls back the application to a previous checkpoint upon a software failure. At a checkpoint, CR stores a snapshot of the application into main memory. Similar to the fork operation, CR copies application memory in a copy-on-write fashion to minimize overhead. By preserving checkpoint states in memory, the overhead associated with slow disk accesses in most previous checkpointing solutions is avoided. This method is also used in previous work [CPL97, JZ88, LNP90, WHV+95, LC98, PLP98, SAKZ04]. Performing a rollback operation is straightforward: simply reinstate the program from the snapshot associated with the specified checkpoint.

Besides memory states, the CR also needs to take care of other system states such as file states during checkpointing and rollback to ensure correct re-execution. To handle file states, CR applies ideas similar to previous work [LC98, SAKZ04] by keeping a copy of each accessed files and file pointers in the beginning of a checkpoint interval and reinstate it for rollback. To simplify implementation, Rx can leverage a versioning file system which automatically takes a file version upon modifications. Similarly, copy-on-write is used to reduce space and time overheads. For some logs file that users may want the old content not to be overwritten during re-execution, Rx can easily provide a special interface that allows applications to indicate what files should not be rolled

back. Other system states such as messages and signals will be described in the next subsection because they may need to be changed to avoid a software bug recurring during re-execution. More details about the lightweight checkpointing method can be found in the previous work [SAKZ04], which uses checkpointing and logging to support deterministic replay for interactive debugging.

In contrast to previous work on rollback and replay, Rx does not require deterministic replay. On the contrary, Rx purposely introduces *nondeterminism* into server's re-execution to avoid the bug that occurred during the first execution. Therefore, the underlying implementation of Rx can be simplified because it does not need to remember when an asynchronous event is delivered to the application in the first execution, how shared memory accesses from multiple threads are interleaved in a multi-processor machine, etc., as what have done in the previous work [SAKZ04].

The CR also supports multiple checkpoints and rollback to any of these checkpoints in case Rx needs to roll back further than the most recent checkpoint in order to avoid the occurring software bug. After rolling back to a checkpoint $CP_i$, all checkpoints which were taken after $CP_i$ are deleted. This ensures that the program does not rollback to a checkpoint which has been rendered obsolete by the rollback process. During a re-execution attempt, new checkpoints may be taken for future recovery needs in case this re-execution attempt successfully avoids the occurring software bug.

**Checkpoint Maintenance**

A possible concern is that maintaining multiple checkpoints could impose a significant space overhead. To address this problem, Rx can write old checkpoints to disks on the background when disks are idle. But rolling back to a checkpoint, which is already stored in disks, is expensive due to slow disk accesses.

Fortunately, Rx does not need to keep too many checkpoints because it strives to bound the recovery time to be 2-competitive as the baseline solution: whole program restarting. In other words, in the worse case, Rx may take twice as much time as the whole program restarting solution (In reality, in most cases as shown in Section 5.6, Rx recovers much faster than the whole program

restart). Therefore, if a whole program restart would take $T$ seconds (This number can be measured by restarting immediately at the first software failure and then be used later), Rx can only repeat rollback/re-execute process for at most $T$ seconds. As a result, Rx cannot rollback to a checkpoint which is too far back in the past, which implies that Rx does not need to keep such checkpoints any more.

More formally, suppose Rx takes checkpoints periodically, let $\tau_1$,$\tau_2$,$\cdots$,$\tau_n$ be the timestamps of the last $n$ checkpoints that have been kept in the *reverse* chronological order. There are two schemes to keep those checkpoints: one is to keep only recent checkpoints, and the other is to keep exponential landmark checkpoints (with $\beta$ as the exponential factor) as in the Elephant file system [SFH+99]. In other words, the two schemes satisfy the following equations, respectively.

$$\tau_i - \tau_{i+1} = \tau_{i-1} - \tau_i \quad (2 \leq i \leq n-1)$$

$$\tau_i - \tau_{i+1} = \beta * (\tau_{i-1} - \tau_i) \quad (2 \leq i \leq n-1)$$

Note that time here refers to application execution time as opposed to elapse time. The latter can be significantly higher, especially when there are many idle periods.

After each checkpoint, Rx estimates whether it is still useful to keep the oldest checkpoint. If not, the oldest checkpoint taken at time $\tau_n$ is deleted from the system to save space. The estimation is done by calculating the worst-case recovery time that requires rolling back to this oldest checkpoint. Suppose after rolling back to a checkpoint, every $ith$ re-execution ($1 \leq i \leq m$) with different environmental changes incurs the overhead $p_i$. Obviously, some environmental changes such as buffer padding impose little time overhead, whereas other changes such as zero-filling buffers incur large overhead. $p_i$s can be measured at run time. Therefore the worst-case recovery time, $RTime$, that requires to roll back to the oldest checkpoint would be (let $\tau$ be the current timestamp):

$$RTime = \sum_{i=1}^{n}\sum_{j=1}^{m}(\tau - \tau_i)(1 + p_j) = \sum_{i=1}^{n}(\tau - \tau_i)\sum_{j=1}^{m}(1 + p_j)$$

If $RTime$ is greater than $T$, the oldest checkpoint taken at time $\tau_n$ is deleted.

### 5.3.3 Environment Wrappers

The environment wrappers perform environmental changes during re-execution for averting failures. Some of the wrappers, such as the memory wrappers, are implemented at user level by intercepting library calls. Others, such as the message wrappers, are implemented in the proxy. Finally, still others, such as the scheduling wrappers, are implemented in the kernel.

**Memory Wrapper** The memory wrapper is implemented by intercepting memory-related library calls such as *malloc()*, *realloc()*, *calloc()*, *free()*, etc to provide environmental changes. During the normal execution, the memory wrapper simply invokes the corresponding standard memory management library calls, which incurs little overhead. During re-execution, the memory wrapper activates the memory-related environmental changes instructed by the control unit. Note that the environmental changes only affect the memory allocation/deallocation made during re-execution.

Specifically, the memory wrapper supports four environmental changes:

(1) Delaying free, which delays recycling of any buffers freed during a re-execution attempt to avoid software bugs such as double free bugs and dangling pointer bugs. A freed buffer is reallocated only when there is no other free memory available or it has been delayed for a threshold of time (process execution time, not elapsed time). Freed buffers are recycled in the order of the time when they are freed. This memory allocation policy is not used in the normal mode because it can increase paging activities.

(2) Padding buffers, which adds two fixed-size paddings to both ends of any memory buffers allocated during re-execution to avoid buffer overflow bugs corrupting useful data. This memory allocation policy is only used in the recovery mode because it wastes memory space.

(3) Allocation isolation, which places all memory buffers allocated during re-execution in an iso-

lated location to avoid corruption useful data due to severe buffer overflow or other general memory corruption bugs. Similar to padding, it is disabled in the normal mode because it has space overhead.

(4) Zero-filling, which zero-fills any buffers allocated during re-execution to reduce the probability of failures caused by uninitialized reads. Obviously, this environmental change needs to be disabled in the normal mode since it imposes time overhead.

*Since none of the above changes violate memory allocation or deallocation interface specifications, they are safe to apply*. At each memory allocation or free, the memory wrapper returns exactly what the application may expect. For example, when an application asks for a memory buffer of size $N$, the memory wrapper returns a buffer with at least size $N$, even though this buffer may have been padded at its both ends, allocated from an isolated location, or zero-filled.

**Message Wrapper** Many concurrency bugs are related to message delivery such as the message order across different connections, the size and number of network packets which comprise a message, etc. Therefore, changing these execution environments during re-execution may be able to avoid an occurring concurrency software bug. This is feasible because servers typically should *not* have any expectation regarding the order of messages from different connections (users), the size and the number of network packets that forms a message, especially the latter two which depend on the TCP/IP settings of both sides.

The message wrapper, which is implemented in the proxy (described in the next subsection), changes the message delivery environment in two ways: (1) It can randomly shuffle the order of the requests among different connections, but keep the order of the requests within each connection in order to maintain any possible dependency among them. (2) It can deliver messages in random-sized packets. Such environmental changes do not impose overhead. Therefore, this message delivery policy can be used in the normal mode, but it does not decrease the probability of the occurrence of a concurrency bug because there is no way to predict in what way a concurrency bug does not occur.

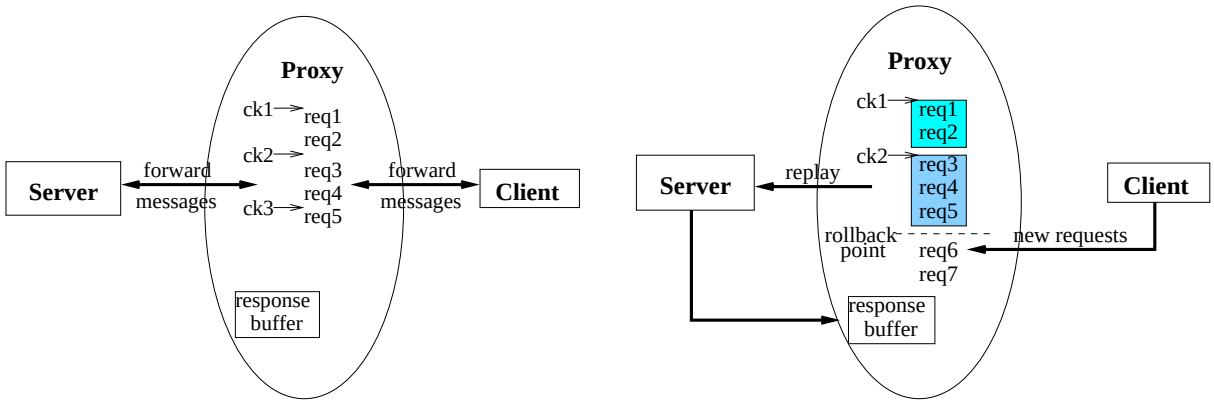
**Process Scheduling** Similarly, concurrency bugs are also related to process scheduling and are

therefore prone to disappear if a different process scheduling is used during re-execution. Rx does this by changing the process' priority, and thus increasing the scheduling time quantum so a process is less likely to switched off in the middle of some unprotected critical region.

**Signal Delivery** Similar to process scheduling, the time when a signal is delivered may also affect the probability of a concurrency bug's occurrence rate. Therefore, Rx can record all signals in a kernel-level table before delivering them. For hardware interrupts, Rx delivers them at randomly selected times, but preserving their order to maintain any possible ordering semantics. For software timer signals, Rx ignores them because during rollback, the related software timer will also be restored. For software exception related signals such as segmentation faults, Rx's sensors receive them as indications of software failures.

**Dropping User Requests** Dropping user requests is a last environmental change before switching to the whole program restart solution. As described earlier, the rationale for doing this is that some software failures are triggered by some malformed requests, either unintentionally or intentionally by malicious users. If Rx drops that request, the server will not experience failure. In this case, the server only denies those dropped requests, but does not affect other requests. The effectiveness of this environmental change is based on our assumption that the client and server use a request/response model, which is generally the case for large varieties of servers including Web Servers, DNS Servers, database servers, etc.

Rx does not need to look for the exact culprit user request. As long as the dropped requests include this request, the server can avoid the software bug and continue providing services. Of course, the percentage of dropped requests should be small (e.g. 10%) to avoid malicious users exploiting it to launch denial of service attacks. Rx can achieve this by performing a binary search on all recently received requests. First, it can drop half of them to see whether the bug still occurs during re-execution. If not, the problem request set becomes one half smaller. If the bug still occurs, it rolls back to drop the other half. If it still does not work, Rx resorts to the whole program restart solution. Otherwise, the binary search continues until the percentage of dropped requests becomes smaller than the specified number. If the percentage upper bound is set to be 10%, it only




**(a) Proxy behavior in normal mode** **(b) Proxy behavior in recovery mode**

Figure 5.3: Proxy in normal and recovery mode. (a) In normal mode, the proxy forward request/response messages between the server and the client, buffers requests, and marks the waiting-for-sending request for each checkpoint (e.g., req3 is marked by checkpoint 2). (b) After the server is rolled back from the rollback-point, as shown in the dashed line to checkpoint 2, the proxy discards the mark of checkpoint 3, replays the necessary requests (req3, req4 and req5) to the server and buffers the incoming requests (req6 and req7). The "unanswered" responses are buffered in the response buffer.

takes 5 iterations of rollback and re-execution.

After Rx finds the small set of requests (less than the specified upper bound) that, once dropped, enable the server to survive the bug, Rx can remember each request's signatures such as the IP address, message size, message MD5 hash value, etc. In subsequent times when a similar bug recurs in the normal mode, Rx can record the signatures again. After several rounds, Rx accumulates enough sets of signatures so that it can use statistical methods to identify the characteristics of those bug-exposing requests. Afterward, if the same bug recurs, Rx can drop only those requests that match these characteristics to speed up the recovery process. In addition, the second level of defense can provide Rx with the exact signature of malicious user requests and facilitate the recovery process.

### 5.3.4 Proxy

The proxy helps a failed server re-execute and makes server-side failure and recovery oblivious to its clients. When a server fails and rolls back to a previous checkpoint, the proxy replays all

the messages received from this checkpoint, along with the message-based environmental changes described in the Section 5.3.3. The proxy runs as a stand-alone process in order to avoid being corrupted by the target server's software defects.

As Figure 5.3 shows, the Rx proxy can be in one of the two modes: *normal* mode for the server's normal execution and *recovery* mode during the server's re-execution. For simplicity, the proxy forwards and replays client messages in the granularity of user requests. Therefore, the proxy needs to separate different requests within a stream of network messages. The proxy does this by plugging in some simple information about the application's communication protocol (e.g. HTTP) so it can parse the header to separate one request from another. In addition, the proxy also uses the protocol information to match a response to the corresponding request to avoid delivering a response to the user twice during re-execution. In the experiments, we have evaluated four server applications, and the proxy uses only 509 lines of code to handle 3 different protocols: HTTP, MySQL message protocol and CVS message protocol.

As shown on Figure 5.3(a), in the normal mode, the proxy simply bridges between the server and its clients. It keeps track of network connections and buffers the request messages between the server and its clients in order to replay them during the server's re-execution. It forwards client messages at request granularity. In other words, the proxy does not forward a partial request to the server. At a checkpoint, the proxy marks the next wait-for-forwarding request in its request buffer. When the server needs to roll back to this checkpoint, the mark indicates the place from which the proxy should replay the requests to the server.

The proxy does not buffer any response in the normal mode except for those partially received responses. This is because after a full response is received, the proxy sends it out to the corresponding client and mark the corresponding request as "answered". Keeping these committed responses is useless because during re-execution the proxy cannot send out another response for the same request. Similarly, the proxy also strives to forward messages to clients at response granularity to reduce the possibility of sending a self-conflicting response during re-execution, which may occur when the first part of the response is generated by the server's normal execution and the second

part of the response is generated by re-execution that may take a different execution path.

However, if the response is too large to be buffered, a partial response is sent first to the corresponding client but the MD5 hash for this partial response is calculated and stored with the request. If a software failure is encountered before the proxy receives the entire response from the server, the proxy needs to check the MD5 hash of the same partial response generated during re-execution. If it does not match with the stored value, the proxy will drop the connection to the corresponding client to avoid sending a self-conflicting response. To handle the case where a checkpoint is taken in the middle of receiving a response from the server, the proxy also marks the exact position of the partially-received response.

As shown on Figure 5.3(b), in the *recovery* mode, the proxy performs three functions to help server recovery. First, it replays to the server those requests received since the checkpoint where the server is rolled back. Second, the proxy introduces message-related environmental changes as described in Section 5.3.3 to avoid some concurrency bugs. Third, the proxy buffers any incoming requests from clients without forwarding them to the server until the server successfully survives the software failure. Doing such makes the server's failure and recovery transparent to clients, especially since Rx has very fast recovery time as shown in Section 5.6. The proxy stays in the recovery mode until the server survives the software failure after one or multiple iterations of rollback and re-execution.

To deal with the output commit problem [SY85] (clients should perceive a consistent behavior of the server despite server failures), Rx first ensures that any previous responses sent to the client are not resent during re-execution. This is achieved by recording for each request whether it has been responded by the server or not. If so, a response made during re-execution is dropped silently. Otherwise, the response generated during re-execution will be temporally buffered until any of the three conditions is met: (1) the server successfully avoids the failure via rollback and re-execution in changed execution environments; (2) the buffer is full; or (3) this re-execution fails again. For the first two cases, the proxy sends the buffered responses to the corresponding clients and the corresponding requests are marked as "answered". Thus, responses generated in subsequent re-

execution attempts will be dropped to ensure that only one response for each request goes to the client. For the last case, the responses are thrown away.

For applications such as on-line shopping that require strict session consistency (i.e. later requests in the same session depend on previous responses), Rx can record the signatures (hash values) of all committed responses for each outstanding session, and perform MD5 hash-based consistency checks during re-execution. If a re-execution attempt generates a response that does not match with a committed response for the same request in an outstanding session, this session can be aborted to avoid confusing users.

The proxy also supports multiple checkpoints. When an old checkpoint is discarded, the proxy discards the marks associated with this checkpoint. If this checkpoint is the oldest one, the proxy also discards all the requests received before the second oldest checkpoint since the server can never roll back to the oldest checkpoint any more.

The space overhead incurred by the proxy is small. It mainly consists of two parts: (1) space for requests received since the undeleted oldest checkpoint, (2) space for "unanswered" responses generated during re-execution in the recovery mode. The first part is small because usually requests are small, and the proxy can also discard the oldest checkpoint to save space as described in Section 5.3.2. The second part has fixed size and can be specified by administrators.

### 5.3.5 Control Unit

The control unit coordinates all the other components in the Rx. It performs three functions: (1) directs the CR to checkpoint the server periodically and requests the CR to roll back the server upon failures. (2) diagnoses an occurring failure based on the symptoms and its accumulated experiences, then decides what environmental changes to apply and where to roll back the server. (3) provides programmers useful failure-related information for postmortem bug analysis.

After several failures, the control unit gradually builds up a failure table to capture the recovery experience for future reference. More specifically, during each re-execution attempt, the control unit records the effects (success or failure) and the corresponding environmental changes into the

table. The control unit assigns a score vector $\langle s_1, s_2, \cdots, s_m \rangle$ to each failure, where $m$ is the number of possible environmental changes. Each element $s_i$ in the vector is the score for each corresponding environmental change $C_i$ for a certain failure. For a *successful* re-execution, the control unit adds one point to all the environmental changes that are applied during this re-execution. For a *failed* re-execution,the control unit subtracts one point from all the applied environmental changes. When a failure happens, the control unit searches the failure table based on failure symptoms, such as type of exceptions, instruction counters, call chains, etc, provided by the Rx sensors. If one table entry matches, it then applies those environmental changes whose scores are larger than a certain threshold $T_s$. Otherwise, it will follow the rules described in Section 5.2 to determine the order how environmental changes should be applied during re-execution. This failure table can be provided to programmers for postmortem bug analysis. It is possible to borrow ideas from machine learning (e.g., a Bayesian classifier) or use some statistical methods as a more "advanced" technique to learn what environmental changes are the best cure for a certain type of failures. Such optimization remains as the future work.

## 5.4 Design and Implementation Issues

**Inter-Server Communication** In many real systems, servers are tiered hierarchically to provide service. For example, a web server is usually linked to an application server, which is then linked to a backend database server. In this case, rolling back one failed server may not be enough to survive a failure because the failure may be caused by its front-end or back-end servers. To address this problem, Rx should be used for all servers in this hierarchy so that it is possible to rollback a subset or all of them in order to survive a failure. Rx can borrow many ideas, such as, coordinated checkpointing, asynchronous recovery, etc, from previous work on supporting fault tolerance in distributed systems [CL99, CL00, EAWJ02, RCL01, AM96], and also from recent work such as micro-reboot [CKF$^+$04]. More specifically, during the normal execution, Rx(s) in the tiered servers take checkpoints coordinately. Once a failure is detected, Rx rolls back the failed server and also

broadcasts its rollback to Rx(s) in other correlated servers, which then roll back correspondingly to recover the whole system. Currently, we have not implemented such support in the Rx and it remains a topic for future study.

**Multi-threaded Process Checkpointing** Taking a checkpoint on a multi-threaded process is particularly challenging because, when Rx needs to take a checkpoint, some threads may be executing system calls or could be blocked inside the kernel waiting for asynchronous events. Capturing the transient state of such threads could easily lead to state inconsistency upon rollback. For example, there can be some kernel locks that have been acquired during checkpoint, and rolling back to such state may cause two processes hold the same kernel locks. Therefore, it is essential to force all the threads to stay at the user level before checkpointing. To do so, Rx sends a signal to all threads, which makes them exit from blocked system calls or waiting events with an EINTR return code. After the checkpoint, the library wrapper in Rx retries the prematurely returned system calls and thus hides the checkpointing process from the target application. This has a bearing on the checkpointing frequency, as a high checkpointing frequency will severely impair the performance of normal I/O system calls, which are likely to be retried multiple times (once at every checkpoint) before long I/Os finish. Therefore, we cannot set the checkpointing interval too small.

**Unavoidable Bug/Failure for Rx** Even though Rx should be able to help servers recover from most software failures caused by common software bugs such as memory corruptions and concurrency bugs, there are still some types of bugs that Rx cannot help the server to avoid via re-execution in changed execution environments. Resource leakage bugs, such as memory leaks, which have accumulative effects on system and may take hours or days to cause system to crash, cannot be avoided by only rolling the server back to a recent checkpoint. Therefore, for resource leaking, Rx resorts to the whole program restart approach because restart can refresh server with plenty of resources. For some of the semantic bugs, Rx may not be effective to avoid them since they may not be related to execution environments. Finally, Rx are not able to avoid the bugs or failures that sensors cannot detect. Solving this problem would require more rigorous dynamic checkers as sensors.

## 5.5 Evaluation Methodology

The experiments described in this section were conducted on two machines with a 2.4GHz Pentium processor, 512KB L2 cache, 1GB of memory, and a 100Mbps Ethernet connection between them. We run servers on one machine and clients on the other. The operating system we modified is the Linux kernel 2.6.10. The Rx proxy is currently implemented at user level for easy debugging. In the future, I plan to move it to the kernel level to improve performance.

We evaluate four different real-world server applications as shown in Table 5.2, including a web server (Apache httpd), a web cache and proxy server (Squid), a database server (MySQL), and a concurrent version control server (CVS). The servers contain various types of bugs, including buffer overflow, data race, double free, dangling pointer, uninitialized read, and stack overflow bugs. Four of them were introduced by the original programmers. We have not yet located server applications which contain uninitialized read or dangling pointer bugs. To evaluate Rx's functionality of handling these two types of bugs, we inject them into Squid separately, renaming the two Squids as Squid-ui (containing an uninitialized read bug) and Squid-dp (containing a dangling pointer bug), respectively.

| App | Ver | Bug | #LOC | App Description |
|---|---|---|---|---|
| MySQL | 4.1.1.a | data race | 588K | a database server |
| Squid | 2.3.s5 | buffer overfbw | 93K | a Web proxy cache server |
| Squid-ui | 2.3.s5 | uninitialized read | | |
| Squid-dp | 2.3.s5 | dangling pointer | | |
| Apache | 2.0.47 | stack overfbw | 283K | a Web server |
| CVS | 1.11.4 | double free | 114K | a version control server |

Table 5.2: Applications and bugs. (App means Application. Ver means Version. LOC means lines of code.)

In this chapter, we design four sets of experiments to evaluate the key aspects of Rx:

- The first set evaluates the functionality of Rx in surviving software failures caused by common software defects by rollback and re-execution with environmental changes. We compare

Rx with whole program restart in terms of client experiences during failure, and in terms of recovery time. In addition, we also compare Rx with the simple rollback and re-execute with *no* environmental changes. This approach is implemented by disabling environmental changes in Rx.

- The second set evaluates the performance overhead of Rx for both server throughput and average response time without bug occurrence. Additionally, we evaluate the space overhead caused by checkpoints and the proxy.

- The third set evaluates how Rx would behave under certain degree of malicious attacks that continuously send bug-expo-sing requests triggering buffer overflow or other software defects. We measure the throughput and average response time under different bug arrival rates. In this set of experiments, we also compare Rx with the whole program restart approach in terms of performance.

- The fourth set evaluates the benefits of Rx's mechanism of learning from previous failure experiences, which are stored in the failure table to speed up recovery.

For all the servers, we implement clients in a similar manner as previous work, such as httperf [MJ98] or WebStone [TS95], sending continuous requests over concurrent connections. For Squid and Apache, the clients spawn 5 threads. Each thread sends out requests to fetch different files whose sizes range in 1KB, 2KB, ..., 512KB with uniform distribution. For CVS, the client exports a 30KB source file. For MySQL, we use two loads. To trigger the data race, the client spawns 5 threads, each of them sending out begin, select, and commit requests on a small table repeatedly. The size of individual requests must be as small as possible to maximize the probability of the race occurring. For the overhead experiments with MySQL, a more realistic load with updates is used. To demonstrate that Rx can avoid server failures, we use another client that sends bug-exposing requests to those servers.

## 5.6 Experimental Results

| Apps | Bugs | Failure Symptoms | Environmental Changes | Clients Experience Failure? | | Recoverable? | | Average Recovery Time (s) | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | Alternatives | Rx | Alternatives | Rx | Restart | Rx |
| Squid | Buffer Overflow | SEGV | Padding | Yes | No | No | Yes | 5.113 | 0.095 |
| MySQL | Data Race | SEGV | Schedule Change | Yes | No | 40% probablity | Yes* | 3.500 | 0.161 |
| Apache | Stack Overflow | Assert | Drop User Request | Yes | No | No | Yes | 1.115 | 0.026 |
| CVS | Double Free | SEGV | Delay Free | Yes | No | No | Yes | 0.010 | 0.017 |
| Squid-ui | Uninit Read | SEGV | Zero All | Yes | No | No | Yes | 5.000 | 0.126 |
| Squid-dp | Dangling Pointer | SEGV | Delay Free | Yes | No | No | Yes | 5.006 | 0.113 |

Table 5.3: Overall results: comparison of Rx and two alternative approaches. The two alternative approaches are whole program restart, and simple rollback and re-execution without environmental changes. The results are obtained by running each experiment 20 times. *The recovery time for the restart approach is measured by having the client not resend the bug-exposing request after reconnection.* Otherwise, the server will crash again immediately after restart. *For MySQL, during the 20 runs, the data race bug never occur during re-execution in Rx after applying various timing-related environmental changes.

### 5.6.1 Overall Results

Table 5.3 demonstrates the overall effectiveness of Rx in surviving failures. For each buggy application, the table shows the type of bug, what symptom was used to detect the bug, and what environmental change was eventually used to avoid the bug. The table also compares Rx to two alternative approaches: the ordinary whole program restart solution and a simple rollback and re-execution without environmental changes. For Rx, the checkpoint intervals in most cases are 200ms except for MySQL and CVS. For MySQL, we use a checkpoint interval of 750ms because too frequent checkpointing causes its data race bug to disappear in the normal mode. The reason for using 50ms as the checkpoint interval for CVS will be explained later when we discuss the recovery time. The average recovery time is the recovery time averaged across multiple bug occurrences in the same execution. Section 5.6.4 will discuss the difference in Rx recovery time between the first time bug occurrence and subsequent bug occurrences.

As shown in Table 5.3, Rx can successfully avoid various types of common software defects, including 5 deterministic memory bugs and 1 concurrency bug. These bugs are avoided during

re-execution because of Rx's environmental changes. For example, by padding buffers allocated during re-execution, Rx can successfully avoid the buffer overflow bug in Squid. Apache survives the stack overflow bug because Rx drops the bug-exposing user request during re-execution. Squid-ui survives the uninitialized read bug because Rx zero-fills all buffers allocated during re-execution. These results indicate that Rx is a viable solution to increase the availability of server applications.

In contrast, the two alternatives, restart and simple rollback/re-execution, cannot successfully recover the three servers (Squid, Apache and CVS) that contain deterministic bugs. For the restart approach, this is because the client notices a disconnection and tries to resend the same bug-exposing request, which causes the server to crash again. For the simple rollback and re-execution approach, once the server rolls back to a previous checkpoint and starts re-execution, the same deterministic bug will occur again, causing the server to crash immediately. These two alternatives have a 40% recovery rate for MySQL that contains a non-deterministic concurrency bug because in 60% cases the same bug-exposing interleaving is used again after restart or rollback. Such results show that these two alternative approaches, even though simple, cannot survive failures caused by many common software defects and thus cannot provide continuous services. The results also indicate that applying environmental changes is the key reason why Rx can survive software failures caused by common software defects, especially *deterministic* bugs.

Because the Rx's proxy hides the server failure and recovery process from its clients, clients do not experience any failures. In contrast, with restart, clients experience failures due to broken network connections. To be fault tolerant, clients need to reconnect to the server and reissue all unreplied requests. With the simple rollback and re-execution, since the server cannot recover from the failure, clients eventually time out and thus experience server failures.

Table 5.3 also shows that Rx provides a significantly better (21-53 times faster) recovery time than restart except for CVS. This is because rollback is a lightweight and fine-grained action due to the in-memory checkpoints. Also, as we find that most faults are detected promptly (usually by crashing), Rx rarely needs to roll back the program further than the recent checkpoint. This minimizes the amount of re-execution necessary. Furthermore, since the program is starting from

a recent execution state, it is unnecessary to initialize data structures or to warm up buffer caches from disks. In contrast, restart is much slower. This is because restart requires the program to be reloaded and reinitialized from the beginning. Any memory state such as buffer caches and data structures need to be warmed up or initialized. Squid is a particularly clear example. For Squid, restart requires 5.113 seconds to recover from a crash, whereas Rx takes only 0.095 seconds. Since the experiments use only a small workload, we expect that, with a real world workload, it will take an even longer time for the whole program restart approach to recovery from failures because it requires a long time to warm up caches and other memory data structures. This result indicates that Rx enables servers to provide highly available services despite common software defects. Instead of experiencing a failure, clients experience an increased response time for a very short period. We expect that after the Rx's proxy is pushed into the kernel, the Rx results will be even better since such optimization will reduce the number of context switches and memory coping overhead.

If the bug-exposing request is not resent after failure, restart has similar recovery time for CVS (otherwise, restart cannot recover the failure for CVS). Restart takes only .01 seconds to recover for CVS, while Rx takes .017 seconds. This is because CVS is implemented using the xinetd daemon as its network monitor. Each connection to CVS causes xinetd to fork and exec a new instance of CVS. Therefore, CVS must have a very low startup time in order to provide adequate performance. Additionally, there is no state shared between different CVS processes except for that of the repository, which is persistently stored on disk. As such, CVS has only minimal state to initialize. Given such a simple application, ordinary restart technique are good enough. For the same reason, even when Rx takes a checkpoint every 50ms, the overhead is still small, less than 11%. But even with such frequent checkpoints, Rx's recovery time is still slightly higher than restart, which indicates for CVS-like servers, restart is a better alternative in terms of recovery time. But note that restart is not failure transparent to clients, and, if the bug-exposing request is resent again by the client after the failure, the same bug (especially deterministic one) is very likely to happen again.

Rx does not hide software defects. Instead, Rx reacts only after a defect is exposed. In addition,

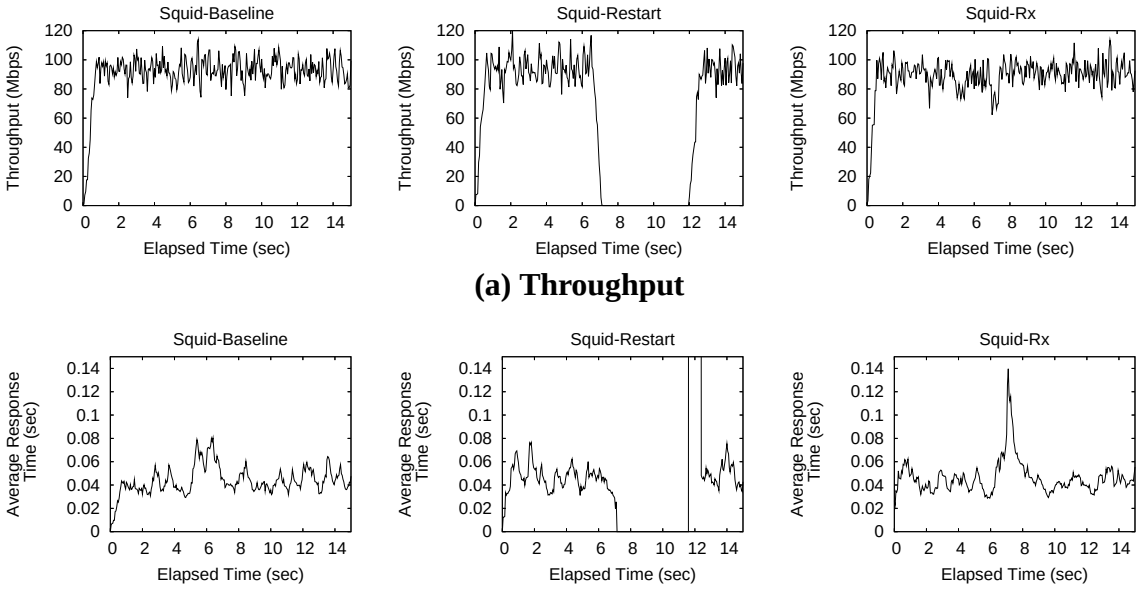
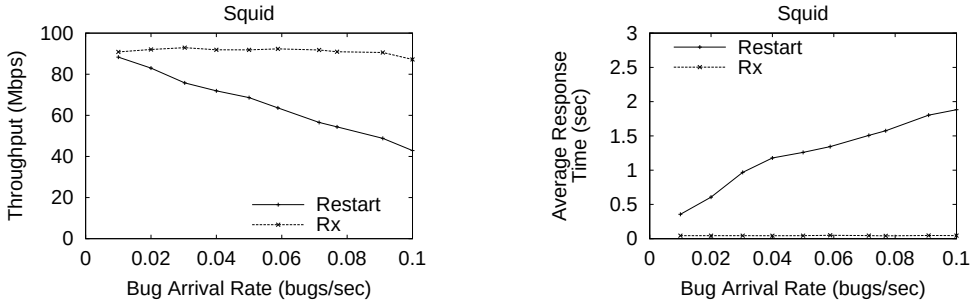



Figure 5.4: Throughput and average response time of Squid. Run squid with Rx and Restart for one bug occurrence (Between time period (7,11.5), there are no measurements for restart because no requests are responded during this period.)

Rx's failure and recovery experiences provide programmers with extra information to diagnose the occurring or occurred software defects. For example, for CVS, Rx is able to avoid the bug by delaying the recycling of recently freed buffers during re-execution. Programmers then should investigate more in the direction of double-free or dangling pointers.

### 5.6.2 Recovery Performance

We have compared Rx with restart in terms of performance during recovery. As shown in Figure 5.4, Rx maintains throughput levels close to that of the baseline case. At the time of bug occurrence (at 7 seconds from the very beginning), the server throughput drops by 33% and the average response time increases by a factor of two for only a very short period of time (17-161 milliseconds). Therefore, a bug occurrence imposes only a small overhead, and has a minimal impact on overall throughput and response time. Restart, on the other hand, has a 5 second period of zero throughput. It services no requests during this period, so there are no measurements for




**(a) Throughput** **(b) Average Response Time**

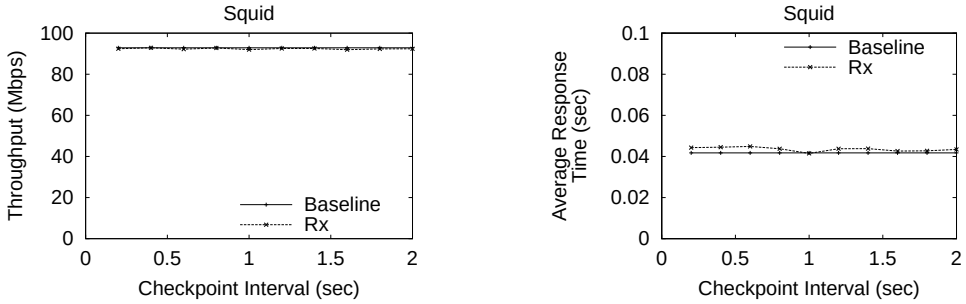
Figure 5.5: Throughput and average response time with different bug arrival rates

response time. Once Squid has restarted, there is a spike in response time because all of the clients get their requests satisfied after a long queuing delay. Because Squid cannot service requests until it has completed the lengthy startup and initialization process, the whole program restart approach significantly degrades the performance upon a failure. Similarly, with a large real-world workload, we expect that the performance with restart will be even worse since the recovery time will become longer and many more requests will be queued, waiting to be serviced.

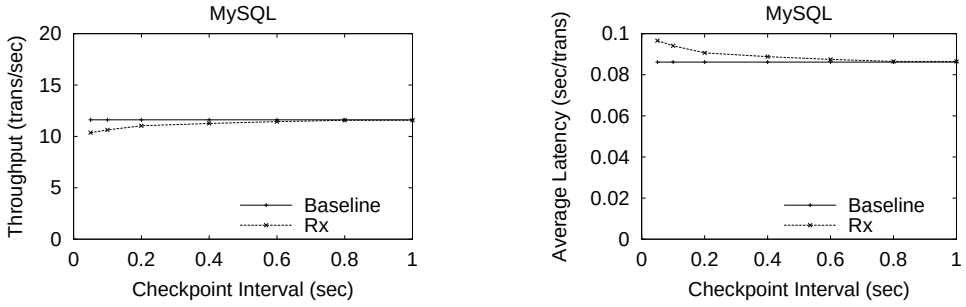
Figure 5.5 further illustrates the Rx's performance in the case of continuous attacks by malicious users who keep issuing bug-exposing requests. The throughput and response time of Rx remain constant as the rate of bug occurrences increases, whereas the performance of restart degrades rapidly. This is because Rx has very small recovery time, while restart spends a long time in recovery. Therefore, if such a bug were triggered by an Internet-wide worm [SPW02] or a malicious user, restart cannot cope. However, since Rx can deal with higher bug arrival rates, Rx can tolerate such attacks much better.

### 5.6.3 Rx Time and Space Overhead

Figure 5.6 shows the overhead of Rx compared to the baseline (without Rx) for various frequencies of checkpointing. The performance of Rx degrades somewhat as the checkpoint interval decreases, but the amount of degradation is small. For squid, both throughput and response time are very close to baseline for all tested checkpoint rates. This is because the network remains the bottleneck for




**(a) Throughput and average response time for Squid**




**(b) Throughput and average response time for MySQL**

Figure 5.6: Rx overhead with different checkpoint intervals The overhead is in terms of throughput and average response time for Squid and MySQL. In these experiments, we do not send the bug-exposing request since we want to compare the pure overhead of Rx with the baseline in normal cases.

all cases. For MySQL, the performance degrades slightly at small checkpoint intervals. Since MySQL is more CPU bound, the additional memory-copying imposed by frequent checkpoints causes some degradation. It is expected that as checkpoints are taken extremely frequently, Rx's overhead will become dominant. However, there is no need for very frequent checkpointing. As shown earlier, even when Rx checkpoints every 200 milliseconds, it is able to provide very good recovery performance. With such a checkpoint interval, the overhead imposed by Rx is quite small, almost negligible for Squid and only 5% for MySQL.

Table 5.4 shows the average memory space overhead of Rx per checkpoint. The space overhead of Rx for each checkpoint is relatively small (45.11-463.60kB). It mainly comes from two parts: updates made during the checkpoint interval and the proxy message buffers. For the first part, Rx uses copy-on-write to reduce space overhead. For the second part, since Rx only records requests in the normal mode and request sizes are usually small, the proxy does not occupy much memory

| Apps | Rx Space Overhead (kB/checkpoint) | | |
|---|---|---|---|
| | kernel | proxy | total |
| Squid | 405.35 | 3.70 | 409.05 |
| Mysql | 300.00 | 0.16 | 300.16 |
| Apache | 460.00 | 3.60 | 463.60 |
| CVS | 42.22 | 2.89 | 45.11 |

Table 5.4: The average space overhead per checkpoint

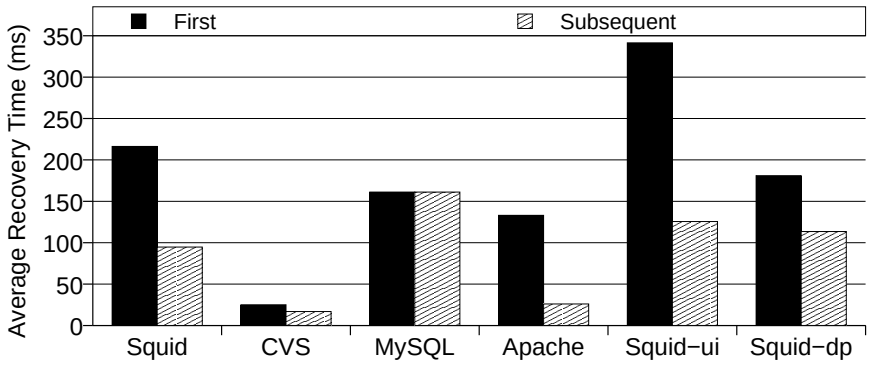



Figure 5.7: Rx recovery time for the first and subsequent bug occurrences

per checkpoint. Therefore, if 2-3MB of space can be used by Rx, Rx is able to maintain 5-20 checkpoints: enough for our recovery purpose.

### 5.6.4 Benefits of the Failure Table

Figure 5.7 reports the server recovery time with Rx when the server encounters the bug for the first time and for subsequent times in the same run. The results show that the failure table can effectively reduce the recovery time when the same bug/failure occurs again. For example, to deal with the first time occurrence of the buffer overflow bug in Squid, Rx applies message reordering, delaying free + message reordering, padding + message reordering sequentially in three consecutive re-execution trials, and finally avoid the bug at the third re-execution. The entire recovery process lasts around 216.7 milliseconds. However, for any subsequent occurrences of the same bug, which can be located in the failure table, Rx applies the correct environmental changes (padding + message reordering) in the first re-execution attempt, thus the recovery time is reduced to 94.7 milliseconds.

For CVS, the failure table also helps to reduce the recovery time from 25 milliseconds to 16.9 milliseconds. For MySQL, the data race bug is avoided at the very first try with message reordering and therefore there is no difference between the first bug occurrence and subsequent ones.

## 5.7 Summary

In summary, Rx is a safe, non-invasive and informative method for quickly surviving software failures caused by common software bugs such as memory corruptions and concurrency bugs and thus providing highly available services. It does so by re-executing the buggy program region in a modified execution environment. It can deal with both deterministic and non-deterministic bugs, and requires few to no modifications to applications' source code. Because Rx does not forcefully change programs' execution by returning speculative values, it introduces no uncertainty or mis-behavior into programs' execution. Moreover, it provides additional feedback to programmers for their bug diagnosis.

The experiments with four server applications that contain six bugs of different types show that Rx can successfully avoid software defects during re-execution and thus provide non-stop services. In contrast, the two tested alternatives, a whole program restart approach and a simple rollback and re-execution without environmental changes, cannot recover the three servers (Squid, Apache and CVS) that contain deterministic bugs, and only have a 40% recovery rate for the server (MySQL) that contains a non-deterministic concurrency bug. These results indicate that applying environmental changes is crucial to survive software failures caused by common software defects, especially deterministic bugs. In addition, Rx also provides fast recovery within 0.017-0.16 seconds, 21-53 times faster than the whole program restart approach for all but one case (CVS). With Rx, clients do not experience any failures except a small increase in the average response time for a very short period of time. To provide such fast recovery, Rx imposes small time and small space overheads.

# Chapter 6

# Conclusions and Future Work

To effectively improve software dependability during production runs, this dissertation proposes multi-level defenses for addressing software bugs with support from the hardware, OS, and runtime. Satisfying the desired characteristics imposed by online detection/surviving systems, the proposed methods incur low runtime overhead, require no source code modification, and have no hardware extension.

In the first-level defense, detecting software bugs once they are triggered, this dissertation proposes a low-overhead, software-only tool, SafeMem, to detect memory leaks and memory corruption bugs, two major forms of software bugs that severely threaten system availability and security. SafeMem makes a novel use of existing ECC memory technology to provide low-overhead, fine-grained memory monitoring functionality to user-level applications. Combined with the proposed intelligent dynamic memory usage behavior analysis, SafeMem can detect memory leaks and corruptions with very low overhead (only 1.6%-14.4% in the experiments) and few to no false positives.

In the second-level defense, detecting exploitation of software bugs if the first-level defense misses them, this dissertation proposes a low-overhead, software-only information flow tracking system, called LIFT, to detect various types of security attacks. Without requiring any hardware changes, LIFT minimizes runtime overhead by exploiting dynamic binary translation and optimizations. More specifically, LIFT aggressively eliminates unnecessary dynamic information flow tracking, coalesces information checks, and efficiently switches between target programs and instrumented information flow tracking code. The experiments show that LIFT effectively detects various types of security attacks and incurs low overhead, only 6.2% for server applications, and

3.6 times on average for seven SPEC INT 2000 applications. The proposed dynamic optimizations effectively reduce the overhead by a factor of 5-12 times.

In the third-level defense, surviving software bugs or failures once reported by the first two level defenses, this dissertation proposes an innovative safe technique, called Rx, to quickly recover programs from many types of software bugs, both deterministic and non-deterministic. The idea, inspired from allergy treatment in real life, is to roll back the program to a recent checkpoint once failure, triggering or exploitation of software bugs that are detected in the first two levels of defenses, and then re-execute the program in a modified environment. The idea is based on the observation that many bugs are correlated with their execution environments, and therefore can be avoided by removing the "allergen" from the environment. The experiments with four server applications that contain six bugs of various types show that Rx can survive all the six software failures and provide transparent fast recovery within 0.017-0.16 seconds, 21-51 times faster than the whole system program restart approach for all but one case (CVS).

This dissertation made contributions to improve software dependability during production runs, a very crucial problem. However, many open problems are left as future work.

One problem is caused by the scalability. Currently the proposed tools work well with single-node systems, but issues still remain open for large-scale distributed systems such as Google search engine. For example, how to identify failed nodes among hundreds of thousands of servers? How to effectively survive failures which are correlated among multiple servers in different hierarchy? To answer these questions, we may borrow ideas like coordinated checkpointing and asynchronous recovery from previous work on supporting fault tolerance in distributed systems.

Rx can potentially tolerate concurrency bugs by manipulating the execution environment. However, how to effectively detect or even prevent the potential concurrency bugs is a difficult and critical problem. Especially with CMP and SMT architectures becoming the mainstream, more and more multi-threaded applications are being written in order to take advantage of the available processors. Consequently, one can expect an increasing number of concurrency bugs in the near future.

This dissertation does not address semantic bugs during production runs. It does not mean there is no such bugs during production runs. A possible research direction is to provide system support for model checking methods to detect hidden semantic bugs during production runs.

Addressing software bugs to improve dependability is part of the answers for the whole dependability problem. Configuration error is another major cause of system failures. How can we use system support to address configuration errors? How to prevent, detect, or tolerate configuration errors? They are interesting and also must-be-solved problems in order to improve software dependability in the real world.

# References


[ABS94] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI)*, pages 290–301, Jun 1994.

[AC77] Algirdas Avizienis and Liming Chen. On the implementation of N-version programming for software fault tolerance during execution. In *Proceedings of the 1st International Computer Software and Applications Conference*, Nov 1977.

[ACZ00] C. Amza, Armando Cox, and W. Zwaenepoel. Data replication strategies for fault tolerance and availability on commodity clusters. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks*, Jun 2000.

[AK98] Paul E. Ammann and John C. Knight. Data diversity: An approach to software fault tolerance. *IEEE Transaction on Computers*, 37(4):418–425, Apr 1998.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactoins on Dependable and Secure Computing*, 1(1):11–33, 2004.

[AM96] Lorenzo Alvisi and Keith Marzullo. Trade-offs in implementing optimal message logging protocols. In *Proceedings of the 15th ACM Symposium on the Principles of Distributed Computing*, May 1996.

[Apa06] Apache http server project. http://httpd.apache.org, May 2006.

[Avi85] Algirdas Avizienis. The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12), 1985.

[Bar81] J. F. Bartlett. A NonStop kernel. In *Proceedings of the 8th Symposium on Operating Systems Principles*, Dec 1981.

[BBG83] Anita Borg, Jim Baumbach, and Sam Glazer. A message system supporting fault tolerance. In *Proceedings of the 9th Symposium on Operating Systems Principles*, Oct 1983.

[BBG+89] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1), 1989.

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent runtime optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, Jun 2000.

[Bei90] B. Beizer. *Software testing techniques (2nd edition)*. Van Nostrand Reinhold Co., 1990.

[Bir96] Kenneth P. Birman. *Building Secure and Reliable Network Applications*, chapter 19. Manning ISBN: 1-884777-29-5, 1996.

[BMMR01] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. In *Proceedings of the SIGPLAN'01 Conference on Programming Language Design and Implementation*, 2001.

[BNG+04] Aniruddha Bohra, Iulian Neamtiu, Pascal Gallard, Florin Sultan, and Liviu Iftode. Remote repair of operating system state using backdoors. In *Proceedings of the 2004 International Conference on Autonomic Computing*, May 2004.

[BPS00] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software : Practice and Experience*, 30(7):775–802, 2000.

[BS96] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, Feb 1996.

[BS98] Andrea Bobbio and Matteo Sereno. Fine grained software rejuvenation models. In *Proceedings of the 1998 International Computer Performance and Dependability Symposium*, Sep 1998.

[BST00] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the USENIX Annual Technical Conference*, 2000.

[BTS99] Arash Baratloo, Timothy Tsai, and Navjot Singh. Libsafe: Protecting critical elements of stacks. White Paper, http://pubs.research.avayalabs.com/pdfs/ALR-2001-019-whpaper.pdf, Dec 1999.

[BWWA06] Edson Borin, Cheng Wang, Youfeng Wu, and Guido Araujo. Software-based transparent and comprehensive control-flow error detection. In *CGO*, 2006.

[CBBKH01] Crispin Cowan, Matt Barringer, Steve Beattie, and Greg Kroah-Hartman. Formatguard: Automatic protection from printf format string vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[CBJW03] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, pages 91–104, Aug 2003.

[CC00] Subhachandra Chandra and Peter M. Chen. Whither generic recovery from application faults? A fault study using open-source software. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks*, Jun 2000.

[CCC+05] Manuel Costa, Joh Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of internet worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, 2005.

[CCF+02] George Candea, James Cutler, Armando Fox, Rushabh Doshi, Priyank Garg, and Rakesh Gowda. Reducing recovery time in a small recursively restartable system. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, Jun 2002.

[CER] CERT/CC. Advisories. http://www.cert.org/advisories/.

[CHM+03a] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. CCured in the real world. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, pages 232–244, Jun 2003.

[CHM+03b] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. CCured in the real world. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, Jun 2003.

[CKF+04] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot – A technique for cheap recovery. In *Proceedings of the 6th Symposium on Operating System Design and Implementation*, Dec 2004.

[CL99] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating System Design and Implementation*, Feb 1999.

[CL00] Miguel Castro and Barbara Liskov. Proactive recovery in a Byzantine-Fault-Tolerant system. In *Proceedings of the 4th Symposium on Operating System Design and Implementation*, Oct 2000.

[CPL97] Y. Chen, James S. Plank, and Kai Li. CLIP: A checkpointing tool for message-passing parallel programs. In *Proceedings of the 1997 ACM/IEEE Supercomputing Conference*, Nov 1997.

[CPM+98] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–78, Jan 1998.

[DD77] Dorothy E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[DDHY92] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *Proceedings of IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[Den76] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[DKA06] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. Enforcing alias analysis for weakly typed languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Jun 2006.

[DLS02] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 57–68, 2002.

[DR03] Brian Demsky and Martin Rinard. Static specification analysis for termination of specification-based data structure repair. In *Proceedings of the 14th IEEE International Symposium on Software Reliability Engineering*, Nov 2003.

[DR05] Brian Demsky and Martin Rinard. Data structure repair using goal-directed reasoning. In *Proceedings of the 2005 International Conference on Software Engineering*, May 2005.

[DRS03] Nurit Dor, Michael Rodeh, and Mooly Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, pages 155–167, Jun 2003.

[EAWJ02] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computer Surveys*, 34(3):375–408, 2002.

[ECCH00] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th Symposium on Operating System Design and Implementation*, 2000.

[ECH+01] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th Symposium on Operating Systems Principles*, 2001.

[EGHT94] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 87–96, Dec 1994.

[Eto06] Hiroaki Etoh. GCC extension for protecting applications from stack-smashing attacks. http://www.trl.ibm.com/projects/security/ssp/, May 2006.

[Far03] Jamil Farshchi. Statistical-based intrusion detection. http://www.securityfocus.com/infocus/1686, Apr 2003.

[FTA02] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, Jun 2002.

[GKIY03] Weining Gu, Zbigniew Kalbarczyk, Ravishankar Iyer, and Zhenyu Yang. Characterization of linux kernel behavior under errors. In *Proceedings of the IEEE Internation Conference on Dependable Systems and Networks*, Jun 2003.

[GMJ+02] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, pages 282–293, Jun 2002.

[God97] Patrice Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, Jun. 1997.

[GPTT97] S. Garg, A. Puliafito, M. Telek, and K. S. Trivedi. On the analysis of software rejuvenation policies. In *Proceedings of the Annual Conference on Computer Assurance*, Jun 1997.

[Gra86] Jim Gray. Why do computers stop and what can be done about it? In *Proceedings of the 5th Symposium on Reliable Distributed Systems*, Jan 1986.

[Gra90] Jim Gray. A census of tandem system availability between 1985 and 1990. Technical report, Tandem Computers Technical Report 90.1, 1990.

[HCXE02] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, pages 69–82, Jun 2002.

[Her05] Steve Herman. Tokyo stock exchange suspends trading after systems failure. Voice of American News, http://www.voanews.com/, Nov 2005.

[HJ92a] R. Hasting and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the USENIX Winter 1992 Technical Conference*, Dec 1992.

[HJ92b] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 125–138, Dec 1992.

[HKKF95] Yennun Huang, Chandra Kintala, Nick Kolettis, and N. Dudley Fulton. Software rejuvenation: Analysis, module and applications. In *Proceedings of the 25th Annual International Symposium on Fault-Tolerant Computing*, Jun 1995.

[HL03] David L. Heine and Monica S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, pages 168–181, Jun 2003.

[HLMSR74] J. Horning, H. Lauer, P. Melliar-Smith, and B. Randell. A program structure for error detection and recovery. *Lecture Notes in Computer Science*, 16:171–187, 1974.

[HOB99] Wei Hua, Jim Ohlund, and Barry Butterklee. Unraveling the mysteries of writing a winsock 2 layered service provider. In *Microsoft Systems Journal*, May 1999.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[HR98] N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 365–377, 1998.

[Int] Intel. Intel e7500 chipset datasheet. http://www.intel.com/design/chipsets/e7500/datashts/290730.htm.

[JK97] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the 3rd International Workshop on Automated and Algorithmic Debugging (AADEBUG)*, pages 13–26, May 1997.

[JZ88] D. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, Aug 1988.

[JZ90] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using optimistic message logging and check-pointing. *Journal of Algorithms*, 11(3):462–491, 1990.

[KBA02] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, 2002.

[LC97] David E. Lowell and Peter M. Chen. Free transactions with rio vista. In *Proceedings of the 16th Symposium on Operating Systems Principles*, Oct 1997.

[LC98] David E. Lowell and Peter M. Chen. Discount checking: Transparent, low-overhead recovery for general applications. Technical report, CSE-TR-410-99, University of Michigan, Jul 1998.

[LCC00] David E. Lowell, Subhachandra Chandra, and Peter M. Chen. Exploring failure transparency and the limits of generic recovery. In *Proceedings of the 4th Symposium on Operating System Design and Implementation*, Oct 2000.

[LCM+05] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, Jun 2005.

[Li88] Kai Li. IVY: A shared virtual memory system for parallel computing. In *Proceedings of the 1988 International Conference on Parallel Processing (ICPP)*, volume II Software, pages 94–101, Aug 1988.

[LLC06] James R. Lyle, Mary T. Laamanen, and Neva M. Carlson. Pest: Programs to evaluate software testing tools and techniques. http://www.itl.nist.gov/div897/sqg/pest/, 2006.

[LLQ+05] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: A benchmark for evaluating bug detection tools. In *2005 Workshop on the Evaluation of Software Defect Detection Tools (Bugs'05)*, Jun 2005.

[LNP90] Kai Li, J. Naughton, and J. Plank. Concurrent real-time checkpoint for parallel programs. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Princiles & Practice of Parallel Programming*, Mar 1990.

[McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publisher, 1993.

[met06] Metasploit project. http://www.metasploit.com/, May 2006.

[Mic] Microsoft. The common language runtime (CLR). http://msdn.microsoft.com/netframework/programming/clr/default.aspx.

[MJ98] David Mosberger and Tai Jin. httperf - a tool for measuring web server performance. *SIGMETRICS Performance Evaluation Review*, 26(3):31–37, 1998.

[ML00] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transaction on Software Engineering and Methodology*, 9(4):410–442, 2000.

[MPC+02] Madanlal Musuvathi, David Park, Andy Chou, Dawson Engler, and David L. Dill. Cmc: A pragmatic approach to model checking real code. In *Proceedings of the 5th Symposium on Operating System Design and Implementation*, Dec 2002.

[MPS+03] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the slammer worm. *IEEE Security and Privacy*, 1(4):33–39, 2003.

[MS00] Evan Marcus and Hal Stern. *Blueprints for High Availability*. John Willey & Sons, 2000.

[Mye99] A. C. Myers. JFLow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[NMW02] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 128–139, Jan 2002.

[NS03] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electroic Notes in Theoretical Computer Science*, 89(2), 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 12th Annual Network and Distributed System Security Symposium*, Feb 2005.

[OGP03] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do internet services fail, and what can be done about it? In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, 2003.

[One96] Aleph One. Smashing the stack for fun and profit. Phrack issue 49 volume 7, 1996.

[PLP98] James S. Plank, Kai Li, and Michael A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, 1998.

[PT03] Milos Prvulovic and Josep Torrellas. Reenact: Using thread-level speculation to debug data races in multithreaded codes. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, Jun 2003.

[QLZ05] Feng Qin, Shan Lu, and Yuanyuan Zhou. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA)*, Feb 2005.

[QTSZ05] Feng Qin, Joe Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating bugs as allergies – a safe method to survive software failure. In *Proceedings of the 20th ACM Symposium on Operating System Principles*, Oct 2005.

[QWL+06] Feng Qin, Cheng Wang, Zhenmin Li, Yuanyuan Zhou, Ho seop Kim, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting general security attacks. In *Proceedings of the 39th Annual IEEE/ACM Internatoinal Symposium on Microarchitecture (to appear)*, Dec 2006.

[Ran75] Brian Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, 1(2):220–232, 1975.

[RCD+04] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee, Jr. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th Symposium on Operating System Design and Implementation*, Dec 2004.

[RCL01] Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov. BASE: Using abstraction to improve fault tolerance. In *Proceedings of the 18th Symposium on Operating Systems Principles*, Oct 2001.

[RL04] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *the 11th Annual Network and Distributed System Security Symposium (NDSS)*, pages 159–169, Feb 2004.

[RLT78] B. Randell, P. A. Lee, and P. C. Treleaven. Reliability issues in computing system design. *ACM Computer Surveys*, 10(2):123–165, 1978.

[Rop94] Marc Roper. *Software Testing*. McGraw-Hill, 1994.

[SAKZ04] Sudarshan Srinivasan, Christopher Andrews, Srikanth Kandula, and Yuanyuan Zhou. Flashback: A light-weight extension for rollback and deterministic replay for software debugging. In *Proceedings of the USENIX 2004 Annual Technical Conference*, Jun 2004.

[sav06] Savant web server. http://savant.sourceforge.net/, May 2006.

[SBB+03] Gregory T. Sullivan, Derek L. Bruening, Iris Baron, Timothy Garnett, and Saman Amarasinghe. Dynamic native optimization of interpreters. In *IVME '03: Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, 2003.

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[SC91] Mark Sullivan and Ram Chillarege. Software defects and their impact on system availability – A study of field failures in operating systems. In *Proceedings of the 21th Annual International Symposium on Fault-Tolerant Computing*, Jun 1991.

[Sco98] Donna Scott. Assessing the costs of application downtime. Gartner Group, May 1998.

[Sco99] D. Scott. Making smart investments to reduce unplanned downtime. Tactical Guidelines Research Note TG-07-4033, Gartner Group, Stanford, CT, Mar 1999.

[SFH+99] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the Elephant file system. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, Dec 1999.

[SFL+94] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 297–306, Oct 1994.

[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 2005 Annual Technical Conference*, Apr 2005.

[SLZD04] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)*, Oct 2004.

[Sno06] Snort - the de facto standard for intrusion detection/prevension. http://www.snort.org, 2006.

[SPW02] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to own the internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium*, Aug 2002.

[sta06] Stack shield – a "stack smashing" technique protection tool for Linux. http://www.angelfire.com/sk/stackshield/, May 2006.

[SY85] Rob Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, 1985.

[Tat06] Simon Tatham. PuTTY: A free Telnet/SSH client. http://www.chiark.greenend.org.uk/˜sgtatham/putty/, May 2006.

[tf800] tf8. Wuftpd: Providing *remote* root since at least1994. http://marc.theaimsgroup.com/?I=bugtraq&m=96171893218000&w=2, Jun 2000.

[Tro06] Mega security. http://www.megasecurity.org/, 2006.

[TS95] G. Trent and M. Sake. Webstone: The first generation in http server benchmarking, 1995.

[US-06] US-CERT. http://www.us-cert.gov/, May 2006.

[VBC+04] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David August. RIFLE:an architectural framework for user-centric information-flow security. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-37)*, Dec 2004.

[VDA+98] W. Vogels, D. Dumitriu, A. Agrawal, T. Chia, and K. Guo. Scalability of the Microsoft Cluster Service. In *Proceedings of the 2nd USENIX Windows NT Symposium*, Aug 1998.

[VDB+98] Werner Vogels, Dan Dumitriu, Ken Birman, Rod Gamache, Mike Massa, Rob Short, John Vert, Joe Barrera, and Jim Gray. The design and architecture of the Microsoft Cluster Service. In *Proceedings of the 28th Annual International Symposium on Fault-Tolerant Computing*, Jun 1998.

[VHBP00] Willem Visser, Klaus Havelund, Guillaume Brat, and Seung-Joon Park. Model checking programs. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, 2000.

[WCA02] E. Witchel, J. Cates, and Krste Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 304–316, Oct 2002.

[WHF93] Yi-Min Wang, Yennun Huang, and W. Kent Fuchs. Progressive retry for software error recovery in distributed systems. In *Proceedings of the 23rd Annual International Symposium on Fault-Tolerant Computing*, Jun 1993.

[WHV+95] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pi-Yu Chung, and Chandra M. R. Kintala. Checkpointing and its applications. In *Proceedings of the 25th Annual International Symposium on Fault-Tolerant Computing*, Jun 1995.

[WK03] John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium*, pages 149–162, Feb 2003.

[XBS06] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium*, Aug 2006.

[XKI03] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent runtime randomization for security. *22nd International Symposium on Reliable Distributed Systems (SRDS'03)*, 00:260, 2003.

[XKPI02] Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel, and Ravishankar K. Iyer. Compiler and architecture support for defense against buffer overflow attacks. In *Proceedings of the 2nd Workshop on Evaluating and Architecturing System Dependability (EASY)*, Jul 2002.

[YTEM04] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file systems errors. In *Processings of the 6th Symposium on Operating Systems Design and Implementation*, 2004.

[ZCL99] Yuanyuan Zhou, Peter M. Chen, and Kai Li. Fast cluster failover using virtual memory-mapped communication. In *Proceedings of the 1999 ACM International Conference on Supercomputing*, Jun 1999.

[ZLF+04] Pin Zhou, Wei Liu, Long Fei, Shan Lu, Feng Qin, Yuanyuan Zhou, Samuel Midkiff, and Josep Torrellas. Accmon: Automatically detecting memory-related bugs via program counter-based invariants. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Micro-architecture (MICRO)*, Dec 2004.

[ZQL+04] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. iWatcher: Efficient architecture support for software debugging. In *Proceedings of the 31st International Symposium on Computer Architecture (ISCA)*, pages 224–237, Jun 2004.

# Vita

## Research Interests

Operating systems; Distributed systems; System and compiler support for software reliability; Software security

## Education

- University of Illinois at Urbana-Champaign

  2006 Ph.D. Computer Science, Adviser: Professor Yuanyuan Zhou
- Institute of Software, Chinese Academy of Sciences

  2001 M.E. Computer Science, Adviser: Yulin Feng
- University of Science and Technology of China

  1998 B.E. Computer Science

## Honors and Awards

- Award paper, the 20th ACM Symposium on Operating Systems Principles, 2005.
- Outstanding teaching assistant award, 2005.
- Micro's Top Picks, 2004.
- Excellent bachelor thesis award, 1998.
- Japanese Telecom scholarship, 1997.

- Zhang Zhongzhi scholarship, 1996.
- First-class scholarship, 1994, 1995.

## RESEARCH EXPERIENCE

- Research Assistant, 2002–2006, University of Illinois, Urbana-Champaign, Champaign, Illinois, USA

  Worked on system support for software dependability.

- Summer Intern, Summer 2005, Intel Research Lab, Beijing, China

  Worked on dynamic information flow tracking systems

- Summer Intern, Summer 2004, NEC Research Lab, Princeton, New Jersey

  Worked on content-addressable storage systems

## TEACHING EXPERIENCE

- Teaching Assistant. Course: Reliable and Robust Software Systems, University of Illinois at Urbana-Champaign, 2005
- Teaching Assistant. Course: Information Storage Systems, University of Illinois at Urbana-Champaign, 2004
- Teaching Assistant. Course: Database Systems, University of Illinois at Urbana-Champaign, 2001

## PUBLICATIONS

Journal Articles

- Rx: System Support for Improving Software Availability

  Feng Qin, Joseph Tucek, and Yuanyuan Zhou. To appear in ACM Transactions on Computer Systems (ACM-TOCS) Special Issue on Best Papers from SOSP'05.
- Efficient and Flexible Architectural Support for Dynamic Monitoring

  Yuanyuan Zhou, Pin Zhou, Feng Qin, Wei Liu and Josep Torrellas. ACM Transactions on Architecture and Code Optimization (ACM-TACO), Vol.2, No.1, Mar. 2005.
- iWatcher: Simple and General Architectural Support for Software Debugging

  Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou and Josep Torrellas. IEEE Micro Special Issue: Micro's Top Picks from Computer Architecture Conferences (Micro's Top Picks), Nov.-Dec. 2004.

Conference Papers

- LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting General Security Attacks

  Feng Qin, Cheng Wang, Zhenmin Li, Yuanyuan Zhou, Ho-seop Kim, and Youfeng Wu. To appear in Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06), Dec. 2006.
- Rx: Treating Bugs as Allergies – A Safe Method to Survive Software Failures

  Feng Qin, Joseph Tucek, Jagadeesan Sundaresan and Yuanyuan Zhou. In Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05), Oct. 2005.
- Treating Bugs as Allergies: A Safe Method for Surviving Software Failures

  Feng Qin, Joseph Tucek and Yuanyuan Zhou. In Proceedings of the USENIX Tenth Workshop on Hot Topics in Operating Systems (HotOS'05), Jun. 2005.
- SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs

  Feng Qin, Shan Lu and Yuanyuan Zhou. In Proceedings of the 10th International Symposium on High-Performance Computer Architecture (HPCA'05), Feb. 2005.

- BugBench: A Benchmark for Evaluating Bug Detection Tools

  Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou and Yuanyuan Zhou. In the 2005 Workshop on the Evaluation of Software Defect Detection Tools (Bugs 2005), Jun. 2005.
- iWatcher: Efficient Architecture Support for Software Debugging

  Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou and Josep Torrellas. In Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA'04), Jun. 2004.
- AccMon: Automatically Detecting Memory-Related Bugs via Program Counter-based Invariants

  Pin Zhou, Wei Liu, Fei Long, Shan Lu, Feng Qin, Yuanyuan Zhou, Sam Midkiff and Josep Torrellas. In Proceedings of 37the Annual IEEE/ACM International Symposium on Microarchitecture (Micro'04), Dec. 2004.
- An OSGI CredentialManager Service

  Jim Basney, Shiva Shankar Chetan, Feng Qin, Sumin Song, Xiao Tu, and Marty Humphrey. In Proceedings of the Workshop on Grid Security Practice and Experience, Jul. 2004.
- Web Application Development: An Object-Oriented Approach

  Bo Zhang, Jing Li, and Feng Qin. In Proceedings of Conference on Software: Theory and Practice (16th IFIP World Computer Congress), Aug. 2000.