Designing Practical Software Bug Detectors Using Commodity Hardware and Common Programming Patterns

Tong Zhang

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Application

Dongyoon Lee, Co-chair
Changhee Jung, Co-chair
Kirk Cameron
Danfeng Yao
Weidong Cui

December 9, 2019
Blacksburg, Virginia

Keywords: Software Bug Detection, Compilers, Commodity Hardware, Data Race Detection, Memory Safety, Permission Check Placement Analysis

Designing Practical Software Bug Detectors Using Commodity Hardware and Common Programming Patterns

Tong Zhang

(ABSTRACT)

Software bugs can cost millions and affect people's daily lives. However, many bug detection tools are not always practical in reality, which hinders their wide adoption. There are three main concerns regarding existing bug detectors: 1) run-time overhead in dynamic bug detectors, 2) space overhead in dynamic bug detectors, and 3) scalability and precision issues in static bug detectors. With those in mind, we propose to: 1) leverage commodity hardware to reduce run-time overhead, 2) reuse metadata maintained by one bug detector to detect other types of bugs, reducing space overhead, and 3) apply programming idioms to static analyses, improving scalability and precision. We demonstrate the effectiveness of three approaches using data race bugs, memory safety bugs, and permission check bugs, respectively. First, we leverage the commodity hardware transactional memory (HTM) selectively to use the dynamic data race detector only if necessary, thereby reducing the overhead from 11.68x to 4.65x. We then present a production-ready data race detector, which only incurs a 2.6% run-time overhead, by using performance monitoring units (PMUs) for online memory access sampling and offline unsampled memory access reconstruction. Second, for memory safety bugs, which are more common than data races, we provide practical temporal memory safety on top of the spatial memory safety of the Intel MPX in a memory-efficient manner without additional hardware support. We achieve this by reusing the existing metadata and checks already available in the Intel MPX-instrumented applications, thereby offering full memory safety at only 36% memory overhead. Finally, we design a scalable and precise function pointer analysis tool leveraging indirect call usage patterns in the Linux kernel. We applied the tool to the detection of permission check bugs; the detector found 14 previously unknown bugs within a limited time budget.

Designing Practical Software Bug Detectors Using Commodity Hardware and Common Programming Patterns

Tong Zhang

(GENERAL AUDIENCE ABSTRACT)

Software bugs have caused many real-world problems, e.g., the 2003 Northeast blackout and the Facebook stock price mismatch. Finding bugs is critical to solving those problems. Unfortunately, many existing bug detectors suffer from high run-time and space overheads as well as scalability and precision issues. In this dissertation, we address the limitations of bug detectors by leveraging commodity hardware and common programming patterns. Particularly, we focus on improving the run-time overhead of dynamic data race detectors, the space overhead of a memory safety bug detector, and the scalability and precision of the Linux kernel permission check bug detector. We first present a data race detector built upon commodity hardware transactional memory that can achieve 7x overhead reduction compared to the state-of-the-art solution (Google's TSAN). We then present a very lightweight sampling-based data race detector which re-purposes performance monitoring hardware features for lightweight sampling and uses a novel offline analysis for better race detection capability. Our result highlights very low overhead (2.6%) with 27.5% detection probability with a sampling period of 10,000. Next, we present a space-efficient temporal memory safety bug detector for a hardware spatial memory safety bug detector, without additional hardware support. According to experimental results, our full memory safety solution incurs only a 36% memory overhead with a 60% run-time overhead. Finally, we present a permission check bug detector for the Linux kernel. This bug detector leverages indirect call usage patterns in the Linux kernel for scalable and precise analysis. As a result, within a limited time budget (scalable), the detector discovered 14 previously unknown bugs (precise).

Dedication

Dedicated to my wife Wei Song, my parents, and my boy Ethan.

Acknowledgments

First of all, I would like to thank my advisors, Drs. Dongyoon Lee and Changhee Jung, and express my highest appreciation and deepest gratitude to them. They are great advisors and mentors. They introduced me to system research, they gave me a lot of guidance, and they shared their profound knowledge with me to make me a better researcher. They care about me and also teach me the wisdom of life to make me a better person. Without their guidance, I won't be able to make such an achievement. I feel very lucky to have Dr. Dongyoon Lee and Dr. Changhee Jung as my advisors and I sincerely hope that we will continue our academic collaborations in the future.

I would also like to thank all committee members and express my sincere gratitude to them, Drs. Kirk Cameron, Danfeng Yao, and Weidong Cui, for their valuable feedback and insightful comment on my research. I am also very grateful for my mentor Drs. Wenbo Shen, Ahemd Azab, and my collaborators Dr. Ruowen Wang, Tongping Liu, as well as Hongyu Liu, and Sam Silvestro for their collaborative efforts in our joint projects.

It was a great pleasure to make a lot of friends in Virginia Tech, Drs. Qingrui Liu, Ke Tian, Zheng Song, Hao Zhang, Xiaokui Shu, Fang Liu, Xiaodong Yu, Bo Li, Run Yu, Yue Cheng as well as Xinwei Fu, Spencer Lee, Peeratham Techapalokul, Dong Chen, Ye Wang, Xuewen Cui, Shengzhe Xu, Peng Peng, Da Zhang, Hang Hu and many other friends. I have so many happy and enjoyable moments with you guys.

I would like to thank my family for the continuous support they have given me throughout my time in graduate school; I could not have done it without their supports.

My thesis is supported in part by National Science Foundation under the grant CCF-1527463, CSR-1750503, CSR-1814430, Google Faculty Research Awards, and Pratt Fellowship.

Contents

List of Figures xiii

List of Tables xvi

1 Introduction 1

1.1 Three Focused Software Bugs 1

1.1.1 Data Race Bugs 2

1.1.2 Memory Safety Bugs 2

1.1.3 Permission Check Bugs 3

1.2 Problem Statements 4

1.2.1 Time and Space Overheads of Dynamic Bug Detectors 5

1.2.2 Scalability and Precision Issues in Static Bug Detectors 6

1.3 Thesis Statement 8

1.4 Contributions 11

1.4.1 Reducing Run-time Overhead of Dynamic Data Race Bug Detectors 11

1.4.2 Reducing Space Overhead of Dynamic Memory Safety Bug Detectors 12

1.4.3 Solving Scalability and Precision Issues in Static Linux Kernel Permission Bug Detectors 13

1.5 Organization 14

2 Literature Review **15**

2.1 Data Races 15

2.1.1 Lockset-based Approaches 16

2.1.2 Overlap-based Approaches 16

2.1.3 Hardware Data Race Detectors 17

2.1.4 Sampling-based Approaches 18

2.1.5 Hybrid Static/Dynamic Approaches 19

2.1.6 Other Approaches to Reduce Dynamic Data Race Detector Overhead 19

2.1.7 Other Related Works 20

2.2 Memory Safety 21

2.2.1 Spatial Memory Safety 21

2.2.2 Temporal Memory Safety 22

2.3 Permission Check Bugs in Linux Kernel 24

2.3.1 Permission Checks in Linux 24

2.3.2 Hook Verification and Placement 26

2.3.3 Kernel Static Analysis Tools 27

2.3.4 Permission Check Analysis Tools 28

3 Efficient Data Race Detection Using Hardware Transactional Memory **30**

3.1 Introduction . . . 31
3.2 Background and Challenges . . . 34
3.2.1 Hardware Transactional Memory . . . 34
3.2.2 Challenges in Using HTM for Race Detection . . . 35
3.3 Overview . . . 36
3.4 Fast Path HTM-based Race Detection . . . 39
3.4.1 Transactionalization . . . 39
3.4.2 Handling Transactional Aborts . . . 41
3.4.3 Optimization . . . 43
3.5 Slow Path Software-based Race Detection . . . 45
3.6 False Negatives . . . 47
3.7 Implementation . . . 48
3.8 Evaluation . . . 50
3.8.1 Methodology . . . 50
3.8.2 Performance Overhead . . . 51
3.8.3 False Negatives . . . 54
3.8.4 Cost-Effectiveness of Data Race Detection . . . 55
3.9 Summary . . . 58

4 Practical Data Race Detection for Production Use 59

4.1 Introduction . . . 60
4.2 Overview . . . 64
4.3 Lightweight Program Tracing . . . 65
4.3.1 PEBS-based Memory Access Sampling . . . 66
4.3.2 PT-based Control-flow Tracing . . . 69
4.3.3 Synchronization Tracing . . . 69
4.4 Recovering Unsampled Memory Accesses . . . 70
4.4.1 Forward Replay . . . 71
4.4.2 Backward Replay . . . 73
4.5 Implementation . . . 75
4.6 Evaluation . . . 76
4.6.1 Methodology . . . 76
4.6.2 Performance Overhead . . . 78
4.6.3 Trace Size . . . 80
4.6.4 Race Detection . . . 82
4.6.5 Memory Operation Reconstruction . . . 83
4.6.6 Offline Analysis Overhead . . . 84
4.7 Summary . . . 85

5 Memory Efficient Temporal Memory Safety Solution for MPX 86

5.1 Introduction . . . 87
5.2 Overview of *BOGO* . . . 89
5.3 *BOGO* Approach Details . . . 93
5.3.1 Hot Bound Table Page Tracking . . . 93
5.3.2 Combine PartialScan and PageFaultScan to Achieve Low Overhead and No False Negative . . . 93
5.3.3 Combine PageFaultScan and RedundancyPredication to Achieve Low Overhead and No False Positive . . . 94
5.4 Optimization . . . 98
5.4.1 No PageFaultScan Optimization . . . 98
5.4.2 FullScan Optimization . . . 99
5.5 Dynamic Adaptation of Queue Size . . . 100
5.5.1 Scan Cost Analysis . . . 100
5.5.2 Impact of HPQ and FAQ Sizes . . . 100
5.5.3 Scan Cost-based HPQ Adaptive Scheme . . . 102
5.6 Discussion . . . 102
5.7 Implementation . . . 103
5.7.1 Spatial Memory Safety . . . 104
5.7.2 Temporal Memory Safety . . . 105
5.8 Evaluation . . . 105

5.8.1 Methodology . . . 105
5.8.2 Security Evaluation . . . 109
5.8.3 SPEC CPU 2006 Benchmark . . . 111
5.8.4 Malloc/Free Benchmark . . . 115
5.8.5 Real-World Applications . . . 116
5.9 Summary . . . 117

6 A Permission Check Analysis Framework for Linux Kernel 118

6.1 Introduction . . . 119
6.2 Examples of Permission Check Errors . . . 122
6.2.1 Capability Permission Check Errors . . . 124
6.2.2 LSM Permission Check Errors . . . 125
6.3 Challenges . . . 126
6.3.1 Indirect Call Analysis in Kernel . . . 126
6.3.2 The Lack of Full Permission Checks, Privileged Functions, and Their Mappings . . . 129
6.4 KIRIN Indirect Call Analysis . . . 130
6.4.1 Indirect Call Target Collection . . . 131
6.4.2 Indirect Callsite Resolution . . . 132
6.5 Design of PeX . . . 134
6.5.1 Call Graph Generation and Partition . . . 135

6.5.2 Permission Check Wrapper Detection 136
6.5.3 Privileged Function Detection 137
6.5.4 Non-privileged Function Filter 138
6.5.5 Permission Check Error Detection 139
6.6 Implementation and Evaluation 140
6.6.1 Evaluation Methodology 141
6.6.2 Evaluation of KIRIN 142
6.6.3 PeX Result 144
6.6.4 Manual Review of Warnings 146
6.6.5 Discussion of Security Bug Findings 147
6.7 Summary 150

7 Conclusion **152**

Bibliography **154**

List of Figures

3.1 TxRace: Transactionalization 37

3.2 TxRace Overview 37

3.3 TxRace Runtime Example 38

3.4 (a) Race detected with long transactions (b) Race missed with short transactions 41

3.5 Detecting data races between fast and slow paths using the strong isolation property of HTM 42

3.6 Tracking the happens-before order of synchronizations on the fast path eliminates false warnings on the slow path 46

3.7 Breakdown of runtime overhead. 49

3.8 Scalability of TxRace 49

3.9 Effectiveness of loop-cut optimization 53

3.10 The number of detected distinct data races across multiple runs for *vips* . . 53

3.11 Cost-Effectiveness of TxRace vs. Sampling 56

3.12 Runtime overhead for *bodytrack* 57

3.13 Recall for *bodytrack* 57

4.1 Overview of the ProRace Architecture. 64

4.2 The vanilla Linux PEBS driver 66

4.3 The ProRace PEBS driver 66

4.4 Forward and Backward Replays. 70

4.5 Example for Forward and Backward Replay 73

4.6 Performance overhead for PARSEC benchmarks 77

4.7 Performance overhead for real applications 77

4.8 Space overhead for PARSEC benchmarks 78

4.9 Space overhead for real applications 78

4.10 Performance overhead comparison 79

4.11 Memory Recovery Ratio 83

4.12 Offline analysis overhead 83

5.1 BOGO: FullScan/PartialScan and PageFaultScan 90

5.2 *BOGO* handler algorithms. 95

5.3 Redundancy prediction: (a) shows a false positive case, and (b) shows how *BOGO* removes the redundant scans and eliminates the false positive. Actions of *BOGO* appears above the bar while the status of HPQ/FAQ does underneath the code. Each color represents a different redundant scan. . . . 96

5.4 Example of sound PartialScan. No further PageFaultScan is required. 98

5.5 MPX compilers overheads: geomean of SPEC 2006. 106

5.6 Performance Overhead. 108

5.7 *BOGO* Performance Overhead Breakdown. 108

5.8 Performance Overhead of Full Memory Safety Solutions. 110

5.9 Memory Overhead. 110

5.10 Sensitivity study: varying HPQ, fixed-size FAQ. 111

5.11 Sensitivity study: varying FAQ, fixed-size HPQ. 111

5.12 Malloc-free benchmark performance. The bar graphs shows the normalized overhead (left y-axis). The line graph shows the free frequency (right y-axis). 115

5.13 Real-world application performance. 115

6.1 Capability check errors discovered by PeX. 123

6.2 LSM check errors discovered by PeX. 125

6.3 Indirect call examples via the VFS kernel interface. 127

6.4 Indirect callsite resolution for `vfs_write`. 132

6.5 Fixing `container_of` missing struct type problem. 133

6.6 PeX static analysis architecture. PeX takes as input kernel source code and permission checks, and reports as output permission check errors. PeX also produces mappings between identified permission checks and privileged functions as output. 134

6.7 Permission check wrapper examples. 137

List of Tables

2.1 Commonly used permission checks in Linux. 24

3.1 TxRace Execution Statistics and Performance. 49

3.2 Cost-Effectiveness of TxRace vs. TSan 56

4.1 Evaluation Setup 76

4.2 Data Race Detection 81

5.1 Impacts of increasing HPQ and FAQ on the number and the cost of Partial, Page Fault, and Full Scans. 98

5.2 llvm-mpx and BOGO validation. 106

5.3 Frequency of Partial, Full, and Page Fault Scans (per second). The sum of Partial and Full Scans represents `free` frequency. 109

5.4 Real-world application test methods. Top four applications are utilities/-clients, while the bottom fives are servers. 115

6.1 Input Statistics for Kernel v4.18.5. 141

6.2 Indirect Call Pointer Analysis. 142

6.3 PeX Results. 144

6.4 Comparison of PeX warnings when used with different indirect call analyses. 144

6.5 Bugs Reported By PeX. **C**onfirmed or **I**gnored. 147

Chapter 1

Introduction

Software bugs are defects in a computer program, that often arise from mistakes and errors in the program source code or design. They make the program produce incorrect results or cause unintended behavior. Unfortunately, some software bugs have resulted in critical real-world disasters, e.g., human deaths [30, 72, 124, 170, 185] and huge financial losses [6, 12, 20, 155, 158, 162]. A study conducted by the National Institute of Standards and Technology in 2002 points out that software bugs cost the US economy an estimated $59 billion in losses every year, or about 0.6% of GDP [194]. In 2018, that number jumped to $1.7 trillion [26], which is about 8% of GDP.

To address software bug problems, we design, develop, and evaluate new practical program analysis tools to help software programmers detect and debug such software bugs, thereby improving software reliability.

1.1 Three Focused Software Bugs

In particular, this thesis focuses on three important classes of software bugs: (1) data race bugs, (2) memory safety bugs, and (3) permission check bugs. The following subsections briefly introduce the three bugs and their real-world impacts.

1.1.1 Data Race Bugs

A data race is a critical software bug that may happen in shared-memory multithreaded programs, which became popular with the advent of multicore systems. In multithreaded programs, the interleavings among threads are nondeterministic, i.e., the program may produce different outputs for a given input across multiple runs. Thus, programmers have to enforce correct order using synchronizations such as locks so that the program behaves as intended. However, this is not a trivial task, leading to data races in many cases.

A *data race bug* happens when the following three conditions are met: (1) two or more threads access the same memory location, (2) at least one of them performs a write operation, and (3) their relative execution order is not explicitly enforced by synchronization primitives such as locks [46, 134, 152]. Such a race condition in multithreaded applications has caused many real-world problems. For example, data races were the root cause of Northeastern US blackout in 2003 [179], which affected around 55 million people, and the mismatched NASDAQ Facebook share prices in 2012 [158], which caused $13 million in losses. Therefore, it is very critical to address data race problems to improve the reliability of multi-threaded programs.

1.1.2 Memory Safety Bugs

Many software programs, such as webservers, databases, and operating systems, are often developed using C/C++ programming languages that allow arbitrary pointer manipulations and unmanaged memory accesses. While this is efficient and flexible, C/C++ programmers are responsible for avoiding invalid memory accesses. If programmers are not careful, C/C++ software can have two forms of memory safety bugs: spatial and temporal memory safety violations.

A *spatial memory safety violation* occurs when a program accesses a memory region beyond the object's designated bound (e.g., buffer overflow or underflow), which may lead to overwriting another object illegally or reading a potentially sensitive data without permission. On the other hand, a *temporal memory safety violation* happens when a program accesses a de-allocated object (e.g., use-after-free or use-after-return).

Unfortunately, many security exploits take advantage of memory safety violations as a first step in seizing control of a program. The first computer worm, Morris [155], exploits a buffer overflow bug in the UNIX finger tool. The infamous Blaster worm [176], which damaged millions of Windows workstations, also exploits a buffer overflow bug. A use-after-free bug in the Darwin kernel is used to jailbreak iOS and achieve local privilege escalation on macOS [107]. Consequently, there is an urgent need to design an efficient and effective tool for detecting such memory safety bugs.

1.1.3 Permission Check Bugs

Access control [174] is an essential security service system software, especially in operating systems. Access control allows only the entities with proper permissions to access and use privileged resources (e.g., files, hardware). In many cases, access control is implemented in the form of permission checks. For example, the DAC [17] in Linux operating systems assigns different permissions for owners, groups, and other users. A write request to a file is first checked against the file permission of that user. The access is granted if the user has proper permissions; otherwise the access is denied.

There are three forms of permission check bugs: missing, inconsistent, and redundant checks. A *missing permission check bug* happens when there is a program path in which a malicious user can access critical resources without going through permission checks. An *inconsistent*

check bug happens when there are multiple program paths to the use of the same privileged resources but they require different permissions, making it hard to determine which one is correct. Finally, a *redundant check bug* occurs when access control unnecessarily checks the same permissions multiple times.

These permission check bugs have caused many security vulnerabilities over the years. For instance, CVE-2011-4080 [3] exposes the Linux kernel message buffer to non-privileged users through a latent path bypassing the permission check in the kernel, which should otherwise deny the access. The message buffer leaks kernel memory mapping information. A more severe and easier-to-exploit bug, CVE-2006-1856 [2], allows an attacker to use `writev`/`readv` syscall in order to bypass intended access restrictions and gain read and write access of security sensitive files. What's more, a permission check bug can even affect cloud service. CVE-2017-17450 [4] enables a non-privileged advisory to escape from a container's namespace isolation and gain the control of system-wide settings.

1.2 Problem Statements

To address the aforementioned problems, researchers and practitioners have designed a variety of dynamic and static program analysis tools. While many of them have been shown to be very useful in detecting data race, memory safety, and permission check bugs, there are three critical challenges still hindering the wide adoption of such tools in practice: (1) run-time overhead in dynamic bug detectors, (2) space overhead in dynamic bug detectors, and (3) scalability and precision issues in static bug detectors.

In this section, we elaborate on these problems and discuss the relevant state-of-the-art solutions for each category. More discussion on other related works is deferred to Chapter 2.

1.2.1 Time and Space Overheads of Dynamic Bug Detectors

Dynamic bug detectors identify a software bug by performing correctness checks along with the program execution at run time. One of the key advantages of using dynamic program analysis is that any reported bug is likely to be an actual bug as it is detected in a lively running context where such an execution is proven to be feasible. In other words, dynamic bug detectors rarely produce false positives, which would require a time-consuming bug validation process. Dynamic bug detectors can also provide ample debugging information (e.g., call stack, branch decision, or variable value), helping developers to reason about the root cause of the detected bug.

However, many existing dynamic bug detectors, especially ones that are designed to detect data races and memory safety bugs (of our interest), incur high run-time performance (time) and memory (space) overheads. Such high overheads make it hard to adopt those tools frequently in a regular software development cycle. What's more, due to stringent quality-of-service requirements, the overhead budget is even more limited in a production environment, making many existing heavyweight tools unusable therein. The following two problem statements (PS) summarize these two overhead problems.

PS 1: Dynamic bug detectors have a high performance overhead for run-time checks.

Additional run-time checks, often performed with instrumented codes, may slow down the original program's execution significantly. In particular, this disadvantage stands out in dynamic data race detection. For instance, Google's ThreadSanitizer, the state-of-the-practice data race detector, may slow down program execution by 30x [180]. FastTrack, the state-of-the-art research prototype, reports an 8.5x slowdown for Java applications [79], and a 57x slowdown for C/C++ applications [75]. Even worser, Intel's Inspector XE, a commercial

product, may incur a 200x performance overhead [173] for some benchmarks. The run-time overhead is also the critical concern for dynamic memory safety bug detectors [192], and we discuss them in detail in Section 2.2.

PS 2: Dynamic bug detectors have a high space overhead for metadata management.

Dynamic bug detectors often need to maintain additional metadata for run-time checks, causing a significant space (memory) overhead. For instance, many memory safety bug detectors keep track of the pointer information. A spatial memory safety solution, [144], maintains upper and lower bounds for each pointer and incurs a 3x space overhead. [199], a temporal memory safety solution, maintains a pointer graph to detect a dangling pointer and incurs 2.4x overhead. Meanwhile, even [181], the most widely used memory safety bug detector, reports a 3.37x memory overhead.

1.2.2 Scalability and Precision Issues in Static Bug Detectors

Unlike dynamic bug detectors, static bug detectors analyze a program's source code and identify potential bugs without running the program. In theory, a static bug detector can detect all the bugs of its kind in a program by exploring all possible program states (with potentially many false positives). Two key merits of static bug detection is that developers have a chance to fix the bugs before deployment, and it does not incur run-time overhead unlike dynamic bug detection.

However, in practice, many static bug detectors suffer from scalability and precision issues. The scalability issue is caused by numerous program states for static tools to explore. That is why static bug detectors often cannot complete an analysis within a limited testing budget, raising practical usability concerns. Besides, static tools make conservative assumptions for

soundness due to the lack of run-time information (e.g., memory states, branch decisions, memory alias information), leading to precision problems. Therefore, static tools often end up producing a large number of false warnings. Unfortunately, false reports require tremendous engineering efforts with no results.

To address the scalability and precision issues in static bug detectors, we are particularly interested in programs with large code bases, such as OS kernels, for which static analysis is a daunting challenge due to these issues. In particular, we focus on a static analysis tool that can detect kernel permission bugs. In the following, we summarize the problem statement (PS) for static bug detectors and discuss them in the context of static analysis tools for the Linux kernel.

PS 3: Static bug detectors often have scalability and precision issues.

The large search space presented by program states makes static bug detectors unscalable. This is especially true for huge and complex codebases such as the Linux kernel (15.8 MLoC). For kernel bug detection, many static analysis tools run their analysis on the call graph of the kernel code. Existing methods are problematic in two ways. The first is the scalability issue. For example, K-Miner [82], a state-of-the-art static memory bug detector for the Linux kernel, integrates a generic inter-procedural pointer analysis tool, SVF [189], for call graph and indirect call analyses. However, there are around 115 thousand static instances of indirect calls in the latest Linux kernel. Even though SVF is optimized using advanced sparse analysis, it cannot handle such a large number of indirect calls. As a workaround, K-Miner applies the SVF analysis only to a small portion of the kernel code at a time, limiting the analysis scope. This brings about the precision issue, i.e., the lack of the whole picture of the kernel code makes the analysis limited (partial), thereby trading precision for scalability.

For the same kernel call graph analysis, some other static kernel bug detectors take a scalable

but imprecise approach. For example, Check-It-Again [207], a Linux kernel lacking-recheck bug detector, uses function type to match indirect call targets, which is imprecise and may lead to wrong call targets that are not feasible at run time. Other recent kernel analysis works like Dr. Checker [133] and Double-Fetch [216] are also using the same imprecise technique to build a kernel call graph for scalability purposes. To produce accurate results within a limited time budget, it is critical for a practical kernel bug detector to be both scalable and precise.

1.3 Thesis Statement

The goal of this thesis is to address the aforementioned limitations in dynamic and static program analysis tools and to design and develop more time- and space-efficient dynamic bug detectors and more scalable and precise static bug detectors. The main contributions of this thesis are summarized as the following thesis statements (TS):

> **TS1**: To address performance overhead problem in dynamic bug detectors (**PS1**), we leverage hardware support available in commodity hardware (originally designed for other purposes) to accelerate dynamic program analysis.

Researchers have proposed custom hardware support to reduce run-time overhead of dynamic bug detectors [37, 66, 210]. Unfortunately, such a research prototype is not readily available on the market. Instead, we begin to look at relevant hardware features in commodity processors. We find out that it is possible to leverage existing hardware features, originally designed for other purposes, to speed up dynamic bug detection. For instance, a processor feature called hardware transactional memory (HTM) [7] was added into Intel's processor to simplify concurrent programming. HTM allows a group of read and write in-

structions to be executed in an atomic manner. We see an opportunity to use HTM to detect data races in (non-TM) multi-threaded programs. In Chapter 3, we demonstrate how HTM can be used to accelerate existing dynamic data race detectors. Furthermore, we also study program tracing and sampling features in many commodity processors. These features were initially designed for low-overhead performance monitoring. We realized they are also useful in implementing a sampling-based lightweight yet effictive data race detector. In Chapter 4, we show how to use program tracing and sampling features for data race detection.

TS2: To address space overhead problems (**PS2**), we propose to reuse metadata maintained by one bug detector to detect another kind of bugs.

In order to perform run-time checks, dynamic bug detectors maintain a collection of their own metadata. When using different bug detectors together, the space overhead simply adds up because their metadata are managed separately. We investigate a method to reuse one set of metadata for multiple purposes. To give an example, Intel MPX [7] is a hardware extension designed to detect spatial memory safety bugs. It maintains bound metadata for each pointer and runs explicit bound checks at the time of pointer dereference to detect a bug. If one would like to detect both spatial and temporal memory safety bugs, the existing option is to run temporal memory safety bug detectors (e.g. DangSan [199]) along with Intel MPX. However, this naïvely combined approach would simply require more memory space for metadata management. To reduce memory overhead for full memory safety, we proposed a solution to reuse Intel MPX's bound metadata to detect temporal memory safety bugs as well, removing the need for maintaining metadata for temporal memory safety. More details are explained in Chapter 5.

TS3: To improve scalability and precision in static bug detectors (**PS3**), we adopt domain-specific programming practices during static program analysis.

Static analyses often make a conservative assumption about program states. To achieve precision, one has to consider numerous program states, raising scalability issues and making it difficult to analyze large-size code. To address scalability issues, one can analyze a small portion of program states, but this will hurt precision. In addition to program source code, domain-specific program practice can be another input for static analysis tools. Domain-specific programming practice can be viewed as a summary of run-time information obtained directly from developers. A generic static analysis tool usually does not consider this knowledge, thus spending more time or making unsound approximations when analyzing the same amount of code. The knowledge of domain-specific program practice can help static analysis tools make correct assumptions about the program faster, avoiding going through complex analysis of large amounts of code. In other words, domain-specific knowledge can help to reduce analysis scope and improve precision by pruning unnecessary search space significantly.

We study common programming patterns used in software development and apply them to design scalable and precise tools. In particular, we study Linux kernel call graph analysis, which is the foundation of many kernel bug detectors. Kernel call graph analysis is tricky because finding the correct target for widely used indirect calls is not trivial. Fortunately, we discovered common usage pattern of indirect calls in the kernel. By applying it, we can not only eliminate false targets but also avoid running a generic pointer analysis tool that is not scalable to the kernel. In Chapter 6, we present a Linux kernel permission bug detector built upon this analysis.

1.4 Contributions

In this thesis, we focus on solving three problems: (**PS1**) the run-time overhead issue of dynamic data race bug detectors, (**PS2**) the space overhead issue of a dynamic memory safety bug detector, and (**PS3**) the scalability and precision issues of a static bug detector for the Linux kernel permission check bug. We summarize our efforts in solving the problems as contributions in the following subsections. We envision that the contributions made in this thesis may be applied to other dynamic and static bug detectors to improve their efficiency.

1.4.1 Reducing Run-time Overhead of Dynamic Data Race Bug Detectors

In **TS1**, to address the run-time overhead problem of dynamic bug detectors, we propose to repurpose commodity hardware features to accelerate dynamic program analysis. We demonstrate this idea on data race detectors and present two dynamic data race bug detectors as follows:

We first present TxRace, a dynamic data race detector using commodity hardware transactional memory (HTM) in Chapter 3. HTM is originally designed to simplify concurrent programming. It executes a bundle of load and store instructions in an atomic manner. HTM abort and rollback transactions once it detects conflict memory accesses of concurrent threads. Our approach runs the program with HTM for fast data race detection and occasionally falls back to a slow but precise software based detection. This design helps us to solve the challenges in designing the HTM-based data race detector, i.e., our design can pinpoint racy instructions, remove false warnings caused by false sharing, and handle non-conflict transaction abort efficiently. We compared our approach with a state-of-the-art

happens-before based data race detector and random sampling-based approach to show its cost-effectiveness.

We then present ProRace, a dynamic data race detector, for production in Chapter 4. Our approach is lightweight, transparent, and effective. Our method reduces overhead by using tracing and sampling features in commodity processors. Those processor features can generate a trace or sample with low overhead and requires no change to program code, minimizing the perturbation to original program execution. Furthermore, our novel forward and backward replay method can reconstruct unsampled memory addresses using the collected trace. Our new driver further lowers the tracing overhead compared to the original Linux kernel driver. Finally, we compared our approach with RaceZ [184], which is another sampling-based dynamic data race detector, using production software such as *Apache* and *MySQL*, and the results show that our approach can detect more bugs at a much lower overhead.

1.4.2 Reducing Space Overhead of Dynamic Memory Safety Bug Detectors

In **TS2**, to address the space overhead problem of dynamic bug detectors, we propose to reuse metadata managed by one bug detector for detecting another type of bugs. We focus on memory safety bug detectors and present a temporal memory bug detector as follows:

In Chapter 5, we present BOGO, a memory efficient temporal memory safety solution (dynamic detector) for Intel's MPX[7], which is a hardware-assisted spatial memory safety solution. Our design does not track its own metadata but instead reuses spatial memory safety check and bound metadata tracked for spatial memory safety. It works by scanning and invalidating bound metadata of freed pointers using an efficient algorithm. The use-after-free bug is detected using the existing spatial memory safety check. Furthermore, without

re-compilation, our design can add temporal memory safety to MPX's spatial memory safety. We also implemented an LLVM-MPX pass to do sound bound checking, which outperforms existing MPX compilers. The evaluation shows that our design can add temporal memory safety with comparable run-time overhead and less memory overhead compared to state-of-the-art solutions.

1.4.3 Solving Scalability and Precision Issues in Static Linux Kernel Permission Bug Detectors

Finally, we study the permission check bug in the Linux kernel and leverage common programming patterns (**TS3**) to address scalability and precision issues in static bug detectors.

We present PeX, a scalable static bug detector for detecting permission check placement errors in the Linux kernel, in Chapter 6. We first solved the problem of scalable and precise inter-procedural control flow graph generation using common indirect call usage pattern in the Linux kernel. Our study shows most of the indirect calls are used to glue drivers and kernel framework; function pointers are usually defined inside a `struct` as an interface. By matching indirect call targets using such an interface, we can achieve scalable and precise indirect call analysis. We show its effectiveness by comparing it with the existing type-based approach and generic pointer analysis-based (SVF [189]) approach. We further automated the process of identifying permission check functions and privileged functions. By running the detector on the latest Linux kernel (v4.18.5), we found 36 new permission check bugs, 14 of which have been confirmed by kernel developers.

1.5 Organization

The remainder of the thesis is organized as follows. Chapter 2 discusses related work for data race bug detection, memory safety bug, and permission check bug. Chapter 3 and Chapter 4 present TxRace and ProRace, data race detectors leveraging commodity hardware transactional memory and PMUs, respectively, to accelerate dynamic data race detection. Chapter 5 details BOGO, a full memory safety solution that detects both spatial and temporal memory safety bugs using Intel MPX's metadata only. Chapter 6 discusses PeX, an efficient and precise permission check analysis framework for the Linux kernel. We conclude in Chapter 7.

Chapter 2

Literature Review

In this chapter, we focus on three kinds of bugs (data race bugs, memory safety bugs, and permission check bugs) and discuss related works. We first discuss existing solutions for data race detection, which includes lockset-based approaches, overlap-based approaches, hardware data race detectors, sampling based approaches, and hybrid static-dynamic approaches. We then discuss existing solutions for memory safety bug detection, which includes spatial memory safety solutions and temporal memory safety solutions. Finally, we discuss different permission checks in the Linux kernel and permission check bugs. We further discuss existing kernel static analysis works, and user application permission check related works.

2.1 Data Races

Data race happens when two or more threads access the same memory location [46, 134, 152], at least one of them is writing to it, and their relative order is not explicitly enforced by synchronization primitives such as locks. Data races have caused many real world problems, e.g, the northeastern blackout [179], mismatched NASDAQ Facebook share prices [158], and security vulnerabilities [219]. Researchers have designed many tools to detect data races. However, such dynamic tools often have too much run-time overhead. We discuss them as follows.

2.1.1 Lockset-based Approaches

Eraser [177] introduced lockset-based approach, which infers data race through violation of locking discipline. As lockset-based algorithms do not consider non-mutex synchronization operations such as conditional variables, they are incomplete (generate false positives) compared to the approach that tracks happens-before order of synchronization operations using vector clocks. FastTrack [79] is known to be the most optimized algorithm in this category, which reduced runtime overhead significantly compared to prior works such as MultiRace [160]. Google's ThreadSanitizer also tracks happens-before order similar to FastTrack. However, high runtime overhead is still a major concern.

2.1.2 Overlap-based Approaches

The recent advances in overlap-based data race detectors [43, 56, 75, 78] have shown their cost-effectiveness and practical benefits by trading soundness for better performance. DataCollider [78] and Kivati [56] use hardware code/data breakpoint mechanism in processors to detect data races. They set a data breakpoint to trap conflicting accesses by other threads. This is similar to conflict detection mechanism in HTM, which detects concurrent conflicting accesses. After setting up the breakpoint, they insert a short amount of time delay to the thread to increase detection probability. The detection window in HTM spans the whole length of a transaction, so the detection probability is likely to be higher than the breakpoint based approach at the cost of false positive. Moreover, the small number of breakpoints (four for x86 hardware) limits its coverage. As another overlap-based detector, IFRit [75] exploits compiler analysis to form interference-free regions where data races can be detected when they overlap. The scope of IFRit may be long in some cases, but it could be very short (e.g., basic block) in other cases since each region may start after the variable is defined in the

SSA form. For performance reasons, IFRit gives up data race detection for those short-scope regions.

2.1.3 Hardware Data Race Detectors

Data race detection also has been the subject of intense research by hardware community. In general, hardware-assisted data-race detectors store metadata (e.g., locksets, vector-clocks) in the cache, piggyback them on coherence protocol messages, and compare them to detect data races.

HARD [235] is a hardware-based implementation of the lockset algorithm, whereas ReEnact [164] and CORD [163] implement happens-before based algorithm in hardware. RADISH [67] proposes hardware-software co-design to enable always-on sound and complete data race detections in which hardware performs the vast majority of race checks and software backs up hardware resource limitations. Conflict Exceptions [131] extends a standard coherence protocol and caches to detect data conflicts between synchronization-free regions. SigRace [143] employs custom hardware address signature where the memory addresses accessed by a processor are hash-encoded and accumulated, and uses it do detect the outcome of potential data races rather than the race itself. Then SigRace relies on checkpointing/rollback to identify actual racing instructions.

RaceTM [88] proposes to use hardware transactional memory in detecting data races. This approach is hardware-only solution that requires additional hardware extension to conventional HTMs like LogTM [139]. For example, RaceTM requires two additional bits (debug read/write bits) for every cache line. They added support to provide the conflict address and responsible racy instructions as well.

Finally, even if TxIntro [126] is not a data race detector, it combines hardware performance

counters (such as HITM, cache miss, and PEBS) with Intel's TSX to infer the conflicting data linear address.

2.1.4 Sampling-based Approaches

LiteRace [135] and Pacer [47] pioneered the use of sampling for reducing the overhead of dynamic data race detection. LiteRace focuses on sampling more accesses in infrequently-exercised code regions, based on the heuristic that for a well-tested application, data races are likely to occur in such a cold region. On the other hand, Pacer uses random sampling and thus its coverage is approximately proportional to the sampling rate used. However, these code instrumentation-based race detectors cause an unaffordable slowdown for some applications, and their detection coverage is limited to the sampled accesses only. For example, though LiteRace shows low 2-4% overhead for Apache, it makes CPU-intensive applications 2.1-2.4x slower, and incurs 1.47x slowdown on average for their tested applications. Similarly, Pacer also reports the average of 1.86x overhead at the 3% sampling frequency.

DataCollider [78] and RaceZ [184] avoid code instrumentation and thus incur a very low overhead, but suffer from low detection coverage. DataCollider [78] makes use of hardware debug breakpoints. After sampling a code/memory location, it sets a data breakpoint and inserts a time delay. A trap during this delay indicates a conflicting access from another thread. Though longer timing delays increase the likelihood of overlapping data races, they also increase the overhead. In addition, hardware restrictions limit the number of concurrently monitored memory locations to four in the latest x86 hardware [101].

RaceZ leverages Intel's PEBS to sample memory accesses. However, due to its reliance on the inefficient Linux PEBS driver, RaceZ has to use a low sampling frequency for performance, thereby compromising the detection coverage. RaceZ also attempts to reconstruct unsampled

memory accesses, but its scope is limited to a single basic block.

2.1.5 Hybrid Static/Dynamic Approaches

Another line of work takes a hybrid static-dynamic approach. RaceMob [111], a recent low-overhead solution, employs static analysis [204] to compute potential data races, and crowdsources runtime race checks across thousands of users. To limit the overhead each user may experience, RaceMob requires a large number of runs to distribute checks, and the number of runs required depends on the precision of the static analysis. Elmas et al. [76], Choi et al. [58] and Chimera [120] are other examples that make use of static data race analysis to reduce runtime cost. In spite of its benefits, static analysis often suffers from precision and scalability issues for large-scale applications, and the recompilation requirement is often not a viable option in production settings.

2.1.6 Other Approaches to Reduce Dynamic Data Race Detector Overhead

Several strategies other than sampling have been explored to reduce the overhead of dynamic data race detection. Overlap-based data race detectors [43, 75, 131] focus on detecting races only when racy instructions or code regions overlap at runtime. Wester et al. [211] parallelizes data race detection. Frost [201] compares multiple replicas of the program running in different schedules.

Greathouse et al. [85] presents a demand-driven race detector. They use hardware performance counters in modern processors to monitor cache events indicating data sharing to turn on race detection. Due to limitation in current hardware, they could identify W→R data

sharing events only, and though all are presumably possible, not all cache sharing causes data races.

Matar et al. [137] exploit Intel TSX (same as ours) to speed up data race detection, but they leveraged HTM simply to replace locks that are used to provide atomicity in metadata updates/checks.

2.1.7 Other Related Works

The idea of using separate low-cost tracing and high-cost (offline) analysis has been used for program runtime monitoring [54, 59], especially in deterministic replay domain [38, 95, 118, 119, 157]. The lowest-overhead solution would be recording only synchronizations and program-input as in RecPlay [171], which guarantees detecting the first race.

There are a large body of works that leverage PMU to reduce the runtime overhead of program monitoring for various purposes. For debugging, Gist [112] uses Intel's PT to track the program execution paths for root cause diagnosis of failures, while CCI [106] uses Intel's Last Branch Record (LBR) to collect the branch trace and the return values for cooperative concurrency bug isolation. For security, FlowGuard [127] uses Intel's PT to achieve transparent and efficient Control Flow Integrity (CFI), while CFIMon [215] uses Intel's Branch Trace Store (BTS) for the same goal. For performance, Brainy [109] leverages Intel's PEBS to understand the effect of the underlying hardware for effective selection of data structures, while Jung et al. [108, 121] use the PEBS to characterize the cache behavior of OpenMP [64] program for dynamic parallelism adaptation.

2.2 Memory Safety

Memory safety bugs are a common source of real-world security vulnerabilities, and they are more popular than data race bugs [5]. Programs written in memory unsafe languages such as C/C++ are often vulnerable to memory safety bugs [192]. Spatial memory safety bugs happen when a memory access does not fall into an object's bound (e.g., buffer overflow). Temporal safety bugs happen when dereferencing using a dangling pointer, which points to a previously a deallocated object (e.g., use-after-free). Existing spatial and temporal memory safety solutions maintain their own metadata in order to detect such a bug. To achieve full memory safety, one can combine a spatial memory bug detector and a temporal memory bug detector. However, this naïve approach would simply double memory overhead. We discuss existing memory safety solutions and their way to maintain metadata as follows.

2.2.1 Spatial Memory Safety

Per-pointer Metadata Solutions

Those solutions maintain bounds information(base and bound) for each pointer. Explicate bound check is performed at the time of pointer dereference. We further distinguish those works into to types: fat-pointer type and dis-joint metadata type. Fat-pointer solutions like [105, 150] augment the pointer and store bounds information adjacent to the original pointer. The downside of fat pointer approaches is that they can change the memory layout thus they can easily break the program relies on the such memory object layout especially it can break binary compatibility. Dis-joint metadata is used by SoftBound[144] and MPX[7]. They maintains bounds information for each pointer in a disjoint memory location thus won't modify memory layout. This makes it binary compatible with existing code. The problem with per-pointer metadata solutions is that when interacting with non-protected

code, pointer metadata won't be maintained, thus making the protection ineffective.

Per-object Metadata Solutions [151, 165, 181, 202, 223] use guard blocks at the beginning and end of memory objects to detect out of bound memory accesses. Those solutions often use page protection mechanism to to detect out of bound access. On the other hand, the use of per-object "red-zone" [181] requires explicit check at the time of pointer dereference to detect out of bound access. [36, 68, 172] allocate disjoint metadata for each memory object and do explicit checking on pointer manipulation. Per-object metadata solutions usually can't guarantee that a pointer can only point to one desired object, but it also allows other objects as well, so there might be false negatives.

2.2.2 Temporal Memory Safety

Special Allocators

Dinakar et al.[69] marks the free'd object page as non-accessible and leverages page protection mechanism to detect memory access through dangling pointers. Cling [35] mitigate the use of dangling pointers by only allowing the address space be used for object with same type and alignment.

Pointer-graph Based Approaches

Nullification [117] achieves temporal safety by maintaining pointer sets and nullifies them at the time of free() which is expensive and may change program behavior for those program leverages dangling pointers but not dereference them. This approach violates C/C++ standard and have high overhead.

Undangle[49] aims at solving temporal memory error by tracking the creation and use of dangling pointer, which also could be very expensive.

DangSan[199] is a temporal memory error detector with enhanced per thread pointer logger data structure to achieve fast on the fly checking.

FreeSentry[225] tracks point by information and invalidate pointers when object is freed.

Identifier-based Schemes

CETS[145] also maintains designated data structure for temporal safety but it is a separated metadata for each pointer and memory object. CETS invalidates the metadata associated with the memory object directly instead of nullifying each pointer. Dangling pointer dereference is thus discovered by explicit check. CETS[145] is added on top of SoftBound to offer temporal safety, the main idea is to add another field to the meta data(key) and allocated memory(lock), which indicate the version information, at the time of dereference, the key and lock is compared for spatial safety.

Watchdog[146] and WatchdogLite[147] take the idea of softbound and CETS to implement ISA extensions to achieve low overhead full memory safety. However these approaches requires new ISA which is not available in current processors.

Other Related Works CFI approaches guarantee program control flow which may be caused by memory corruption attack. Stackguard[62] aims at preventing control flow hijack by using on stack buffer overflow attack. ARM Pointer Authentication uses signature for pointers to prevent crafted pointers by malicious program from being dereferenced. This technique can mitigate the memory safety issue but can not fully prevent it from happening. Finally, Sok[192] is a comprehensive survey concluding prior work which tries to tackle memory safety problem.

Table 2.1: Commonly used permission checks in Linux.

Type	***Total #***	***Permission Checks***
DAC	3	generic_permission, sb_permission, inode_permission
Capabilities	3	capable, ns_capable, avc_has_perm_noaudit
LSM	190	security_inode_readlink, security_file_ioctl, etc..

2.3 Permission Check Bugs in Linux Kernel

Previously we discussed dynamic detect data races detectors and dynamic memory safety bugs detectors. In this subsection, we focus on another important category of bug detectors, static bug detectors. We are especially interested in static bug detectors dealing with large codebase, the Linux kernel. We depict deficiencies of existing kernel analysis works in the following subsection, and we focus our discussion on Linux kernel permission checks [174], which is an essential security enforcement scheme in operating systems. They assign users (or processes) different access rights, called permissions, and enforce that only those who have appropriate permissions can access critical resources (e.g., files, sockets). In the kernel, access control is often implemented in the form of *permission checks* before the use of *privileged functions* accessing the critical resources. Without proper permission check function placement, adversaries can take advantage of this bug to bypass security check and gain access to critical resources [32].

2.3.1 Permission Checks in Linux

This section introduces DAC, Capabilities, and LSM in Linux kernel. Table 2.1 lists practically-known permission checks in Linux. Unfortunately, the full set is not well-documented.

Discretionary Access Control (DAC)

DAC restricts the accesses to critical resources based on the identity of subjects or the

group to which they belong [166, 198]. In Linux, each user is assigned a user identifier (uid) and a group identifier (gid). Correspondingly, each file has properties including the owner, the group, the `rwx` (read, write, and execute) permission bits for the owner, the group, and all other users. When a process wants to access a file, DAC grants the access permissions based on the process's uid, gid as well as the file's permission bits. For example in Linux, `inode_permission` (as listed in Table 2.1) is often used to check the permissions of the current process on a given inode. More precisely speaking, however, it is a wrapper of `posix_acl_permission`, which performs the actual check.

In a sense, DAC is a coarse-grained access control model. Under the Linux DAC design, the "root" bypasses all permission checks. This motivates fine-grained access control scheme—such as Capabilities—to reduce the attack surface.

Capabilities

Capabilities, since Linux kernel v2.2 (1999), enable a fine-grained access control by dividing the root privilege into small sets. As an example, for users with the `CAP_NET_ADMIN` capability, kernel allows them to use `ping`, without the need to grant the full root privilege. Currently, Linux kernel v4.18.5 supports 38 Capabilities including `CAP_NET_ADMIN`, `CAP_SYS_ADMIN`, and so on. Functions `capable` and `ns_capable` are the most commonly used permission checks for Capabilities (as listed in Table 2.1). Both determine whether a process has a particular capability or not, while `ns_capable` performs an additional check against a given user namespace. They internally use `security_capable` as the basic permission check.

Capabilities are supposed to be fine-grained and distinct [16]. However, due to the lack of clear scope definitions, the choice of specific Capability for protecting a privileged function has been made based on kernel developers' own understanding in practice. Unfortunately, this leads to frequent use of `CAP_SYS_ADMIN` (451 out of 1167, more than 38%), and it is

just treated as yet another root [15]; grsecurity points out that 19 Capabilities are indeed equivalent to the full root [14].

Linux Security Module (LSM)

LSM [214], introduced in kernel v2.6 (2003), provides a set of fine-grained pluggable hooks that are placed at various security-critical points across the kernel. System administrators can register customized permission checking callbacks to the LSM hooks so as to enforce diverse security policies. The latest Linux kernel v4.18.5 defines 190 LSM hooks. One common use of LSM is to implement Mandatory Access Control (MAC) [23] in Linux (e.g., SELinux [186, 187], AppArmor [11]). MAC enforces more strict and non-overridable access control policies, controlled by system administrators. For example, when a process tries to read the file path of a symbolic link, `security_inode_readlink` is invoked to check whether the process has `read` permission to the symlink file. The SELinux callback of this hook checks if a policy rule can grant this permission (e.g., `allow domain_a type_b:lnk_file read`). It is worth noting that the effectiveness of LSM and its MAC mechanisms highly depend on whether the hooks are placed *correctly* and *soundly* at all security-critical points. If a hook is missing at any critical point, there is no way for MAC to enforce a permission check.

2.3.2 Hook Verification and Placement

There is a series of studies on checking kernel LSM hooks. Automated LSM hook verification work [234] verifies the complete mediation of LSM hooks relying manually specified security rules. While [80] automates LSM hook placements utilizing manually written specification of security sensitive operations. However, required manual processes are error-prone when applied to huge Linux code base. Edwards et al. [74] proposed to use dynamic analysis to detect LSM hook inconsistencies. While PeX is using static analysis, can achieve better code

coverage.

AutoISES [193] is the most closely related work to PeX. AutoISES regards data structures, such as the structure fields and global variables, as privileged, applies static analysis to extract security check usage patterns, and validates the protections to these data structures. The difference between AutoISES and PeX is three-fold. First, PeX is privileged function oriented while AutoISES is more like data structure oriented. Second, AutoISES is designed for LSM only, whose permission checks (hooks) are clearly defined, and therefore it is not applicable to DAC and Capabilities due to their various permission check wrappers. In contrast, PeX works for all three types of permission checks. Third, AutoISES uses type-based pointer analysis to resolve indirect calls, while PeX uses KIRIN to resolve indirect calls in a more precise manner.

There are also works[81, 141, 142] that extend authorization hook analysis to user space programs, including X server and postgresql. However, their approaches canot be applied to the huge kernel scale. Moreover, all of above works either do not analyze indirect calls at all, or apply over approximate indirect call analysis techniques, such as type-based approach or field insensitive approach. To the contrary, PeX uses KIRIN, a precise and scalable indirect call analysis technique, which can also benefit these works by finding more accurate indirect call targets.

2.3.3 Kernel Static Analysis Tools

Coccinelle [156] is a tool that detects a bug of pre-defined pattern based on text pattern matching on the symbolic representation of bug cases. This is basically intra-procedural analysis. Building upon Coccinelle, Wang et al. proposed another pattern matching based static tool which detects potential double-fetch vulnerabilities in the Linux kernel [206].

Sparse [27] is designed to detect the problematic use of pointers belonging to different address space (kernel space or userspace). Later, Sparse was used to build a static analysis framework called **Smatch** [25] for detecting different sorts of kernel bugs. However, Smatch is also based on intra-procedural analysis, thus it can only find shallow bugs.

Double-Fetch [216], **Check-it-again** [207] focus on detecting time of check to time of use (TOCTTOU) bugs. **Dr. Checker** [133] is designed for analyzing Linux kernel drivers. It adopts the modular design, allowing new bug detectors to be plug-in easily. **KINT** [208] applies taint analysis to detect integer errors in Linux kernel while **UniSan** [128] leverages the same analysis to detect uninitialized kernel memory leakages to the userspace. **Chucky** [217] also uses a taint analysis to analyze missing checks in different sources in userspace programs and Linux kernel. However, Chucky can handle only kernel file system code due to the lack of pointer analysis. Note that to resolve indirect call targets, all these works leverage a type-based approach, which is not as accurate as KIRIN, thus suffering from false positives.

MECA [218] is an annotation based static analysis framework, and it can detect security rule violations in Linux. **APISan** [228] aims at finding API misuse. It figures out the right API usage through the analysis of existing code base and performs intra-procedural analysis to find bugs. To achieve the former, APISan relies on relaxed symbolic execution which is complementary to the techniques used in PeX.

2.3.4 Permission Check Analysis Tools

Engler et al. propose to use programmer beliefs to automatically extract checking information from the source code. They apply the checking information to detect missing checks [77]. **RoleCast** [188] leverages software engineering patterns to detect missing security checks in web applications. **TESLA** [39] implements temporal assertions based on LLVM instrument,

in which the FreeBSD hooks are checked by inserted assertions dynamically. Different from TESLA, PeX uses KIRIN to analyze jump targets of all kernel function pointers statically, achieving better resolution rate and code coverage. **JIGSAW** [203] is a system that can automatically derive programmer expectations on resources access and enforce it on the deployment. It is designed for analyzing userspace programs, cannot be applied to kernel directly.

JUXTA [138] is a tool designed for detecting semantic bugs in filesystem while **PScout** [40] is a static analysis tool for validating Android permission checking mechanisms. **Kratos** [183] is a static security check framework designed for the Android framework. It builds a call graph using LLVM and tries to discover inconsistent check paths in the framework. However, Android has well-documented permission check specifications [9], i.e., privileged functions and the permission required for them are both clearly defined. In contrast, the Linux kernel has no such documentation, which makes it impossible to apply PScout and Kratos to Linux kernel permission checks.

Chapter 3

Efficient Data Race Detection Using Hardware Transactional Memory

Detecting data races is important for debugging shared-memory multithreaded programs, but the high runtime overhead prevents the wide use of dynamic data race detectors. In this chapter, we presents TxRace, a new software data race detector that leverages commodity hardware transactional memory (HTM) to speed up dynamic data race detector (**PS1**). TxRace instruments a multithreaded program to transform synchronization-free regions into transactions, and exploits the conflict detection mechanism of HTM for lightweight data race detection at runtime. However, the limitations of the current best-effort commodity HTMs expose several challenges in using them for data race detection: (1) lack of ability to pinpoint racy instructions, (2) false positives caused by cache line granularity of conflict detection, and (3) transactional aborts for non-conflict reasons (e.g., capacity or unknown). To overcome these challenges, TxRace performs lightweight HTM-based data race detection at first, and occasionally switches to slow yet precise data race detection only for the small fraction of execution intervals in which potential races are reported by HTM. According to the experimental results, TxRace reduces the average runtime overhead of dynamic data race detection from 11.68x to 4.65x with only a small number of false negatives.

3.1 Introduction

A data race occurs when two or more threads access the same memory location, at least one of them is a write, and their relative order is not explicitly enforced by synchronization primitives such as locks [46, 134, 152].

Data races often lie at the root of other concurrency bugs such as unintended sharing, atomicity violation, and order violation [130]. There are many real-world examples showing the severity of data races, including the northeastern blackout [179], mismatched Nasdaq Facebook share prices [158], and security vulnerabilities [219]. Moreover, data races make it difficult to reason about the possible behaviors of programs. The C/C++11 standards [46, 102, 103] give no semantics to programs with data races, and the data race semantics for Java programs [134] is considered to be too complex [205].

To address this problem, a variety of dynamic data race detectors [76, 79, 98, 160, 180, 227] have been proposed to help programmers write more reliable multithreaded programs. However, such dynamic tools often add too much runtime overhead. For example, FastTrack, a state-of-the-art happens-before based detector, incurs a 8.5x slowdown for Java programs [79] and a 57x slowdown for C/C++ programs [75]. For different set of benchmarks, Intel's Inspector XE incurs a 200x slowdown [173], and Google's ThreadSanitizer incurs a 30x slowdown [180]. Such high overhead hinders the widespread use of dynamic data race detectors in practice, despite their good detection precision.

We designed TxRace (Chapter 3), a novel dynamic data race detector which leverages hardware transactional memory to accelerate data race detection.

Transactional Memory (TM) [94] was proposed to simplify concurrent programming as a new programming paradigm, and hardware support for transactional memory has recently become available in commodity processors such as Intel's Haswell processor [96, 97]. This work

exploits the observation that the conflict detection mechanism of HTM can be repurposed for lightweight data race detection in conventional multithreaded programs not originally developed for TM.

However, naively leveraging HTM does not automatically guarantee efficient data race detection. Rather, the limitations of the current commodity HTMs expose three challenges in using them for data race detection: (1) As the HTMs are designed to transparently guarantee atomicity and isolation between concurrent transactions, they do not provide a way to pinpoint racy instructions or conflicting addresses; (2) Since data conflicts are detected at the cache line granularity, false alarms could be reported due to the false sharing; and (3) Due to the best-effort nature of existing commodity HTMs, transactions may abort for reasons other than data conflicts, including exceeding the hardware capacity, interrupts and exceptions.

To overcome these challenges, it first instruments a multithreaded program to transform all code regions between synchronizations (including critical sections) into transactions (Figure 3.1). At runtime, it then performs a two-phase data race detection comprising fast and slow paths. During the initial fast path, it detects potential data races using the low-overhead data conflict detection mechanism of HTM. In this stage, the detected races are only potential races, as the conflict might be due to false sharing in the cache line. When a data conflict is detected, the current HTMs do not identify the instruction that caused the transaction to abort, the conflicting address, or the other conflicting transaction. it addresses this problem by artificially aborting all the (in-flight) concurrent transactions, rolling back them to the state before the data conflict occurred, and then performing software-based *sound* (no false negative) and *complete* (no false positive) data race detection [79, 98, 180] for the concurrent code regions. This work refers to the rollback and subsequent re-execution with software-based data race detection as the slow path, which enables it not only to pinpoint

racy instructions but also to filter out any false positives. it also relies on the slow path to cover the code regions, which cannot be monitored by transactions due to the limitations of existing commodity HTMs. This conservative approach reduces the chance of missing data races at the cost of runtime overhead, and it includes an optimization technique to avoid repeated capacity aborts.

The experimental results show that using Intel's Restricted Transactional Memory (RTM) and Google's ThreadSanitizer (TSan) for the fast and slow paths respectively, it achieves runtime overhead reduction of dynamic data race detection from 11.68x (TSan) to 4.65x (this work) on average. Using an HTM-based detector during the fast path may lead to missing data races if they do not overlap in concurrent transactions (and for other reasons).

Nevertheless, it incurs only a few false negatives (high recall of 0.95) for tested applications.

This work makes the following contributions:

• To the best of our knowledge, TxRace is the first software scheme that demonstrates how commodity hardware transactional memory can be used to build a lightweight dynamic data race detector.

• TxRace proposes novel solutions to address the challenge in designing HTM-based data race detector. They enable TxRace to pinpoint racy instructions, remove false data race warnings caused by false sharing, and handle non-conflict transactional aborts efficiently.

• The paper presents experimental results showing cost effectiveness of TxRace compared to a state-of-the-art happens-before based data race detector and its random sampling based approach.

3.2 Background and Challenges

This section briefly introduces hardware transactional memory systems and discusses the challenges in using them in data race detection.

3.2.1 Hardware Transactional Memory

Transactional Memory (TM) [94] provides programmers with transparent support to execute a group of memory operations in a user-defined transaction in an atomic (all or nothing) and isolated (the partial state of a transaction is hidden from others) manner. Hardware support for transactional memory has been implemented in IBM's Blue Gene/Q supercomputers [92] and System z mainframes [104]; Sun's (canceled) Rock processor [71]; Azul's Vega processor [60]. Recently, HTM has become available in commodity processors used in desktops such as Intel's Haswell processor [96, 97].

TxRace leverages Intel's Transactional Synchronization Extensions (TSX) introduced in the Haswell processors. Intel TSX includes Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM) supports, where the latter enables general-purpose transactional memory. Intel RTM provides a new set of instructions comprising *xbegin*, *xend*, *xabort*, and *xtest* to help programmers initiate, complete, abort a transaction, and check its status, respectively. Intel RTM uses the first level (L1) data cache to track transactional states, and leverages the cache coherence protocol to detect transactional conflicts [83, 224]. Intel RTM supports *strong isolation*, which guarantees transactional semantics between transactions and non-transactional code [136]. For conflict management, Intel RTM uses a simple *requester-wins* policy in which on a conflicting request, the requester always succeeds and the conflicting transactions abort [45].

3.2.2 Challenges in Using HTM for Race Detection

At first glance, it might be expected that HTMs can trivially provide lightweight data race detection. However, the commercial HTMs, including Intel RTM, share limitations that hinder their adoption for data race detection.

First, though HTMs can detect the presence of data conflicts and abort, HTMs including Intel RTM do not provide programmers with the problematic instructions that caused the transaction to abort, or with the affected memory addresses. Moreover, the concurrent transactions to which the competing instructions belong may have been successfully committed according to TM semantics. This implies that programmers cannot reason about the pairs of memory accesses involved in the data race.

Second, HTMs detect data conflict by leveraging a cache coherence mechanism. Conflicts are therefore discovered at the cache-line granularity (64-bytes in the Intel Haswell processor). This may produce false warnings in data race detection due to non-conflicting operations on variables that share a cache line. By comparison, traditional dynamic data race detectors identify data races at the word (or byte) granularity, significantly reducing the likelihood of false positives.

Third, HTMs have bounded resources, and irrevocable I/O operations are not supported. Intel RTM does not support arbitrarily long transactions, simply aborting any transactions exceeding the capacity of the hardware buffer for transactional states [93, 169]. Moreover, changing privilege level always forces a transaction abort. The implication is that a transaction should not include any system call.

Finally, a transaction in best-effort (non-ideal) commodity HTMs including Intel RTM may be aborted for an unknown reason (neither due to data conflict nor due to capacity overflow). Intel's reference manuals [96, 97] illustrate some causes of unknown aborts, such as operating

system context switches for interrupt or exception handling. However, neither the exact abort reason nor the problematic instruction is provided to programmers, which makes it hard to work around such a transaction abort.

3.3 Overview

TxRace is a lightweight software dynamic data race detector that leverages hardware transactional memory support in modern commodity processors. The key insight is that TxRace can detect potential data races with a very low runtime overhead by initially relying on the conflict detection mechanism of HTM. The detected races are potential because the data conflict between transactions might be caused by false sharing in the same cache line. When transactions commit without data conflicts, TxRace incurs only the small runtime overhead of the transactional execution. In this sense, this HTM-based check is called *fast path* in which TxRace first takes to quickly identify potential data races. To address the aforementioned challenges of HTM-based detectors, on a data conflict, TxRace artificially aborts all concurrent (in-flight) transactions, rolls them back to the state before the data conflict occurs, and performs software-based sound and complete data race detection in an on-demand manner. Such re-execution with software-based detection is called *slow path*, which allows TxRace not only to pinpoint racy instructions but also to filter out false positives caused by false sharing.

Figure 3.2 shows an overview of TxRace. It consists of a compile-time instrumentation and a two-phase data race detection at runtime. TxRace inserts fast path transactional codes (e.g., *xbegin*, *xend*) and slow path sound and complete data race detection codes (e.g.,FastTrack [79], ThreadSanitizer [180]) into the original program at compile-time. Then, TxRace makes use of the two-phase data race detection at runtime. This allows TxRace to

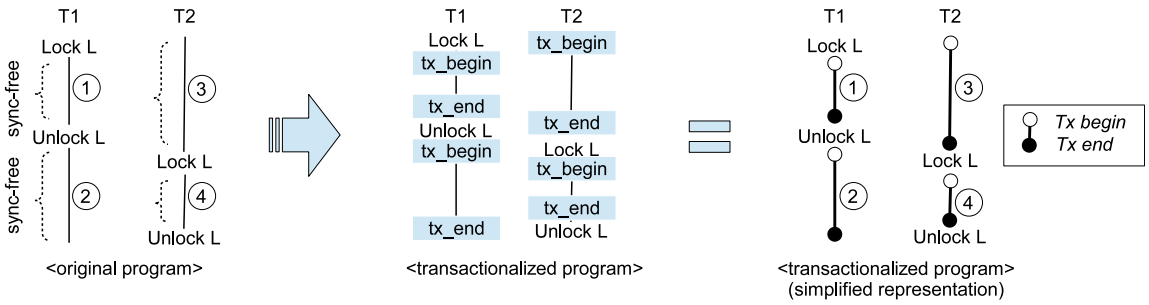


Figure 3.1: TxRace: Transactionalization

selectively perform sound and complete (but slow) data race detection for only a small fraction of the whole execution, leading to significant runtime overhead reduction compared to a traditional dynamic data race detection.

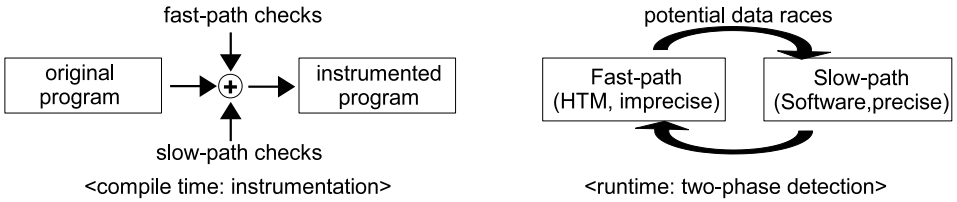


Figure 3.2: TxRace Overview

As illustrated in Figure 3.1, TxRace transforms program regions between synchronizations (*synchronization-free regions*) into transactions. Then, TxRace performs HTM-based race detection between program regions that overlap in parallel at runtime. For example, for an execution where program regions ① and ③ overlap, TxRace checks potential data races between those two regions. Similarly, program regions between ② and ③; or between ② and ④ are checked when they run concurrently. On the other hand, the original synchronization lock L prevents the program regions ① and ④ (and corresponding transactions) from being overlapped at runtime. Therefore, the HTM will never observe conflicting memory accesses in critical sections protected by the same lock. In the following figures, a white circle corresponds to the beginning of a transaction, a black circle to its end.

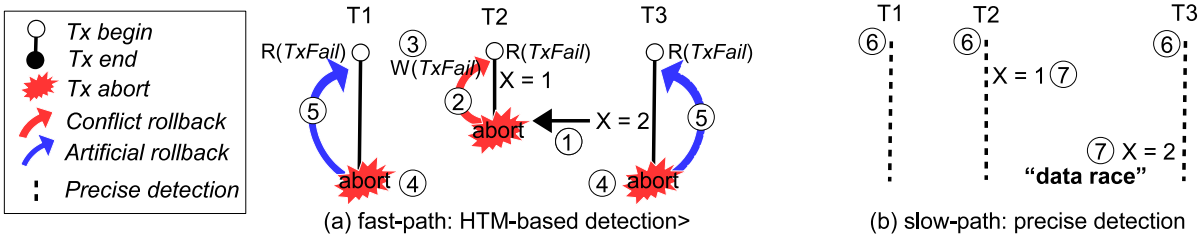


Figure 3.3: TxRace Runtime Example

After compile-time instrumentation, TxRace detects data races at runtime using the fast and slow paths as follows. Figure 3.3 shows an example with three threads, T1, T2, and T3, each performing one transaction. Suppose the shared variable X is not protected by the common lock, so that the transactions of T2 and T3 may run concurrently. During the fast path, each transaction begins by first reading a shared global flag named *TxFail* (as a part of instrumented code together with *xbegin*). Then, TxRace relies on HTM to detect data conflicts. Suppose a transaction in T3 causes another transaction in T2 to abort by accessing the shared variable X (step ①). Intel RTM employs a *requester-wins* conflict resolution strategy in which on a conflicting request, the requester always succeeds and the conflicting transactions abort [45]. Thus, the concurrent transactions of T1 (no conflict) and T3 (winner) may proceed further. To pinpoint the precise data race condition, TxRace immediately aborts the in-flight transactions by making the aborted transaction in T2 update *TxFail* (step ③) right after its rollback. Intel RTM supports *strong isolation* that guarantees transactional semantics between transactions and non-transactional code [136]. Together with the requester-wins policy, the strong isolation property in Intel RTM cause a transaction to abort if there is a conflicting access from a non-transaction code. Therefore, the update to *TxFail* causes all the concurrent transactions to abort artificially (step ④) as they have read *TxFail* at the beginning of the transaction. When all the concurrent transactions are rolled back (step ⑤), they resume execution on the slow path in which HTM is no longer used (step ⑥), but software-based sound and complete data race detection

is performed (step ⑦) instead. When the slow path finishes for the program regions where potential data races are detected, TxRace switches back to the fast path in which HTM is used for the next program regions.

3.4 Fast Path HTM-based Race Detection

This section first describes how TxRace instruments original programs to detect potential data races as well as how it handles different types of transactional aborts, and then discuss optimization techniques used for reducing performance overhead.

3.4.1 Transactionalization

To exploit HTM for potential data race detection, TxRace transforms a code region between synchronization operations (including a critical section) into a transaction as illustrated in Figure 3.1. To be specific, at compile-time, TxRace inserts transaction begin instructions (*xbegin*) at thread entry points and after synchronization operations; and transaction end instructions (*xend*) at thread exit points and before synchronization operations. System calls require special consideration due to HTM limitations. Intel RTM, for example, aborts a transaction if a change in privilege level takes place. Consequently, TxRace ends the current transaction prior to each system call and begins a new transaction immediately after the system call in order to guarantee forward progress.

Furthermore, TxRace instruments each transaction to read the shared flag *TxFail* immediately after *xbegin*. As discussed in Section 3.3, Intel RTM uses the *requester-wins* policy that allows the requester to succeed on a conflicting request and aborts the conflicting transactions [45]; and supports *strong isolation* that guarantees transactional semantics be-

tween transactions and non-transactional code [136]. These two properties cause an in-flight transaction to abort on a conflicting access from non-transaction code. As TxRace makes transactions read *TxFail* when they start, when a data conflict is detected, the aborted transaction can artificially abort other in-flight transactions by writing to the flag *TxFail*. For example in Figure 3.3, when the aborted transaction in T2 writes to *TxFail*, the concurrent in-flight transactions in T1 and T3 which have read *TxFail* get aborted. Similar techniques have been used to abort in-flight transactions in hybrid TM systems [51, 65, 123] or to enable transactional lock elision [34].

HTMs can only detect those data races that result in conflicts between concurrent transactions. This suggests that maximizing the size of the transactions inserted at compile-time will minimize the likelihood of false negatives. It would be ideal to transform each synchronization-free region into a single transaction. However, as mentioned above, TxRace cuts transactions across systems calls inevitably. A performance optimization called *loop-cut* discussed further in Section 3.4.3 may also lead to cut a transaction originally formed for a synchronization-free region.

Figure 3.4 (a) and (b) show that the length of transactions can affect the detection of data races. Both (a) and (b) show unsynchronized and potentially-concurrent writes to the shared variable X from threads T1 and T2: a race condition. In (a), each thread executes a single lengthy transaction. The long transaction length increases the likelihood that the two transactions will overlap, and in this case the data race on X is detected. In (b), each thread executes two transactions, and the data race on X is more likely to be missed (a false negative) as a result. As shown in (b), suppose the first transaction in T1 includes *X=1* and successfully commits prior to the beginning of the second transaction in T2, which includes *X=2*. However, if the two transactions do not overlap, then neither will abort. For this reason, similar to other overlap-based data race detectors [43, 56, 75, 78], TxRace may miss

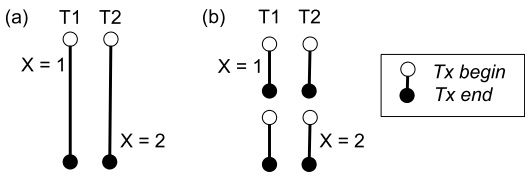


Figure 3.4: (a) Race detected with long transactions (b) Race missed with short transactions

data races if they happen far apart in time. Though perfect soundness is thus out of reach, the experimental results in Section 3.8 show that TxRace trades only a few false negatives (recall of 0.95) for excellent performance.

3.4.2 Handling Transactional Aborts

The best-effort Intel RTM does not guarantee that a transaction will eventually commit and make progress. In addition to data conflicts, there are many architectural and micro-architectural conditions that may cause a transaction to abort. When a transaction is aborted, Intel RTM rolls back the transaction to the point where it begins and reports the abort type(s) in the register. TxRace handles transaction aborts according to the abort reason as follows, while juggling the competing goals of reducing false negatives, decreasing overhead, and offering a forward progress guarantee.

Conflict. A transaction aborted due to a data conflict indicates a potential data race. To conduct precise data race detection, TxRace updates the shared flag (named *TxFail*) that every transaction begins by reading, forcing all the concurrent (in-flight) transactions to abort and roll back (Figure 3.3). TxRace then performs slow path software-based sound and complete data race detection among the code regions that overlapped with the aborted code region. Once a potential data race is detected by the fast path, the software-based slow-path detector will winnow out the false positives and to find data races if one exists.

Retry. A transaction aborted with "retry" status might succeed if retried. If this flag is set

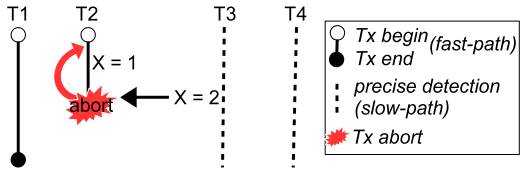


Figure 3.5: Detecting data races between fast and slow paths using the strong isolation property of HTM

in conjunction with the conflict flag described above, TxRace treats the case as a conflict and follows the slow path. Otherwise, TxRace retries the transaction.

Capacity. When a transaction is aborted due to overflow, TxRace makes only the thread that observed the capacity abort fall back to slow path. Unlike the case for data conflicts, TxRace does not artificially abort the other concurrent transactions (by not updating the shared flag *TxFail*) since there is no indication of a potential data race with concurrent transactions. Using concurrent slow and fast path executions minimizes performance overhead while still giving TxRace high detection coverage (fewer false negatives) and the guarantee of forward progress. Figure 3.5 demonstrates how TxRace can detect data races when both fast and slow paths run at the same time. Here, threads T1 and T2 are on the fast path, while threads T3 and T4 are on the slow path due to capacity aborts. In this case, data races between threads T1 and T2 can be detected using HTM-based fast-path detection, and data races between thread T3 and T4 can be detected using precise slow-path detection. The interesting case is a race condition between fast path and slow path threads. If T2 is on the fast path and T3 is on the slow path as shown in the example, the strong isolation property of Intel RTM ensures that the transaction in T2 will be aborted in the event that T3 makes the conflicting access to the variable X that T2 has accessed. In this case, TxRace handles the conflict abort as described above. Because T3 is already in the slow path, the precise data race condition can be identified once TxRace puts T2 in the slow path.

Unknown. A transaction may abort with an unknown (unspecified) reason. As TxRace

enforces that a transaction does not include a system call (Section 3.4.1), this is most likely due to unexpected operating system context switches to handle interrupts, exceptions, etc. To guarantee forward progress while achieving high detection coverage (fewer false negatives), TxRace treats this case the same as the capacity abort.

Debug/Nested. The debug bit is set when a transaction aborts upon encountering a debug breakpoint, while the nested bit is set when a transaction was aborted during a nested transaction. Neither of these conditions may happen as a result of the TxRace transactionalization process; no debug breakpoints are used, and TxRace does not introduce nested transactions. TxRace simply ignores this case.

3.4.3 Optimization

To reduce performance overhead at runtime, TxRace applies several optimizations in the fast path. First, TxRace checks if the program is in the single-threaded mode or not (e.g., in the very beginning of program execution before spawning child threads). If so, there should be no races, thus TxRace simply does not use HTM to monitor the program execution to avoid unnecessary cost of transactions. If a function is profiled to be invoked in both single-threaded and multi-threaded modes, then at compile-time TxRace clones the function and instruments only the version that is called in multithreaded mode.

Second, Txrace reuses the same static analysis algorithm that Google TSan uses to avoid unnecessary data race checks. If a memory operation is statically proven to be data race free, then TSan does not instrument it. TxRace also does not insert transaction codes for those code regions that are not instrumented by Google TSan to hook memory accesses for data race detection.

Third, for regions containing a small number of memory operations, the overhead associated

with HTMs exceeds the cost of the software-based slow path. If a code region contains fewer than K memory operations, TxRace favors the slow path. In our experiment we chose $K = 5$.

Finally, TxRace leverages our so-called *loop-cut* optimization for transactions that includes loops with a large number of iterations; these loops are a frequent cause of capacity aborts. By default, TxRace falls back to the slow path when a transaction experience a capacity abort to obtain better detection capability for those code regions at the cost of some performance overhead. To reduce this overhead, the loop-cut optimization aims to end the long transaction before prior to a capacity abort. TxRace first profiles an application with representative input to identify the candidate loops for the loop-cut optimization. In this study, TxRace leverages the Last Branch Recorder (LBR), a branch tracing facility in Intel processors [86], which allows TxRace to identify the last branch taken before a transaction aborts. Then, TxRace inserts the following loop-cut logic to the end of the candidate loop body.

The high level idea is for TxRace to keep track of the number of loop iteration (called *loop-cut-threshold*). When the transaction experiences a capacity abort in the loop, TxRace takes the slow path at first (default behavior), but when the same loop is executed next time, TxRace uses this loop-cut-threshold to cut the current transaction early in the middle of loop iterations, placing the rest of the iterations into another transaction to avoid capacity aborts. As discussed above, short transactions cut by loop-cut optimization may lead to false negatives.

HTM semantics make this implementation slightly tricky. Note that as the loop is a part of transaction, it is not possible to use a counter incremented per loop iteration to obtain the precise loop-cut-threshold value; updates to the counter will not survive a transaction abort. TxRace addresses this problem by setting a small initial estimate loop-cut-threshold (two

in our experiment) and by incrementing and decrementing the estimate when the transaction commits/aborts, respectively (outside the transaction). This approach enables TxRace to estimate the last largest loop-cut-threshold allowing the transaction to commit. This work calls this scheme, which dynamically learns the loop-cut-threshold at runtime, *TxRace-DynLoopcut*.

As another scheme, *TxRace-ProfLoopcut* profiles an application with representative input to figure out the initial loop-cut-threshold value beforehand. This approach allows TxRace to avoid even the very first capacity abort. Similar to *TxRace-DynLoopcut*, *TxRace-ProfLoopcut* handles misprofiling by adjusting the threshold when the transaction commits or aborts accordingly. Section 3.8.2 evaluates the effectiveness of the two loop-cut optimization schemes.

3.5 Slow Path Software-based Race Detection

For software-based data race detection during the slow path, TxRace uses Google's ThreadSanitizer (TSan) [173, 180], an open-source state-of-practice data race detector. Similar to the well-known sound and complete FastTrack algorithm [79], TSan keeps track of the happens-before order for each memory location using a shadow memory. Then, TSan detects data races when accesses to shared locations are not ordered. This process requires instrumenting (1) synchronization operations to track the happens-before order; and (2) memory operations to look up shadow memory and compare their happens-before order. By design, TSan is complete (no false positive). However, to bound memory overhead, TSan maintains N (default 4) shadow cells per 8 application bytes, and replaces one random shadow cell when all shadow cells are filled. This may affect soundness (no false negative) of data race detection. Thus, this work configured TSan to have enough number of shadow cells to be sound as well.

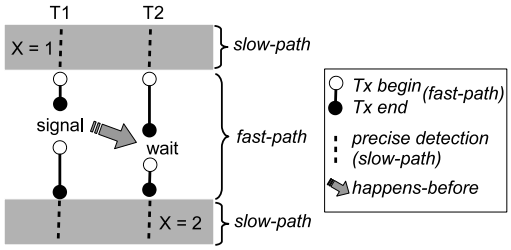


Figure 3.6: Tracking the happens-before order of synchronizations on the fast path eliminates false warnings on the slow path

The potential interplay of fast and slow path threads necessitates additional overhead during the fast path. A naive fast path could rely solely on HTM to detect potential data races, obviating the need to track the happens-before order imposed by synchronization operations. However, threads may alternate between fast and slow paths for precise data race detection. When TSan finishes in the slow path, TxRace resumes the use of the fast path to monitor the next regions, achieving better performance. This design requires TxRace to keep track of the happens-before order of synchronization operations even during the fast path to remove false warnings during slow path. Figure 3.6 describes this feature in more detail. Suppose that threads T1 and T2 have executed the slow path, fast path, and slow path in turn for some reason other than conflicting accesses to X, and that there was a happens-before order between *signal* and *wait* that appeared during the fast path. If TxRace does not track this happens-before order during the fast path, the slow path data race detector would report a data race between $X{=}1$ and $X{=}2$, which is a false warning. The performance overhead breakdown in Section 3.8.2 shows that tracking synchronization operations is not that expensive during the fast path.

3.6 False Negatives

TxRace is complete (no false positive) but unsound (some false negative). There are four main reasons why TxRace could miss data races. First, fast-path HTMs do not detect data conflicts between transactions that do not overlap in time, as discussed in Section 3.4.1. This is different from sound (happens-before based) data race detectors such as FastTrack or Google's TSan, which identify races by tracking the happens-before order of synchronization operations. In this sense, TxRace resembles overlap-based data race detectors [43, 56, 75, 78].

Second, when a transaction is aborted due to data conflict and TxRace writes the shared flag *TxFail* to abort others, there is no guarantee that some of the already-running transactions will not commit before they see the write. In this case, even though TxRace triggers the slow path, the race will not occur again and thus cannot be detected.

Third, race detection between the fast and slow paths (Figure 3.5) only works in one direction. If the slow path thread makes a shared memory access *before* the fast path thread makes a conflicting access, then the HTM's strong isolation guarantee does not apply. As a result, when the opposite of the situation in Figure 3.5 happen, TxRace will not trigger the slow path, and the race will not be detected.

Finally, TxRace by nature shares the limitations of the underlying HTM system. As of now, Intel Haswell processor does not support more concurrent transactions than the total number of hardware threads available. This implies that the number of threads that can be monitored by HTM during fast path is limited.

During evaluation, the thread count was restricted to be smaller than the hardware thread counts, ruling out the forth reason. All of the observed false negatives were due to non-overlapping transactions.

3.7 Implementation

TxRace instrumentation framework is implemented in the LLVM compiler framework [115]. As a very first process, we translate application source codes into LLVM IR (Intermediate Representation) and perform instrumentation as a custom transformation pass. During this process, we do not include external libraries such as standard *libc*, *libc++*, *libm*, etc., assuming that such libraries are thread safe, and users are interested in detecting data races in application codes. External libraries may be included into our scope when their LLVM IR is provided. It is worthwhile to note that we included all the internal libraries that are provided with core application codes such as *gsl*, *libjpeg*, *glib*, *libxml2*, etc. in PARSEC benchmark suite [41] into our analysis.

Instrumentation for fast path needs to intercept synchronization operations and the program points before/after system calls so that transaction begin/end codes can be inserted. For example, a new transactional region starts after a new thread starts or after each system call. As we do not include standard C/C++ libraries into our scope, we instrument system calls at the library call boundary; i.e., before and after calls to library functions that may invoke system calls such as synchronization (e.g., *PThread* library); standard I/O (e.g., *read*, *write*); and dynamic memory management library (e.g., *malloc*, *free*). For the third party libraries whose source codes are not available, dynamic binary instrumentation tools [48, 132, 151] can be used to profile the program with representative input and to identify a list of external library functions invoking system calls. Misprofiling would result in unknown aborts caused by undetected system calls. TxRace falls back to slow path in case of unknown aborts (by default), thus misprofiling only adds runtime overhead, and does not harm detection coverage.

For slow path, we use off-the-shelf Google's ThreadSanitizer (TSan) [173, 180], an open-

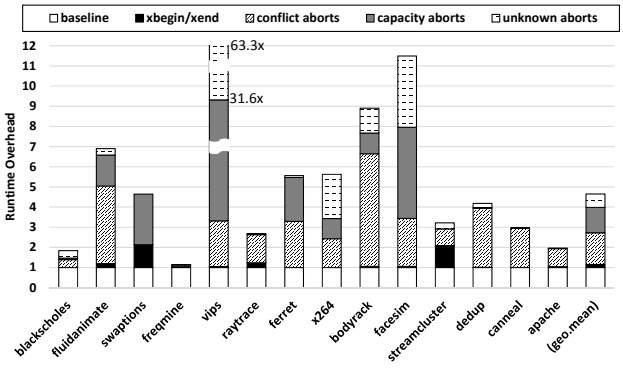


Figure 3.7: Breakdown of runtime overhead.

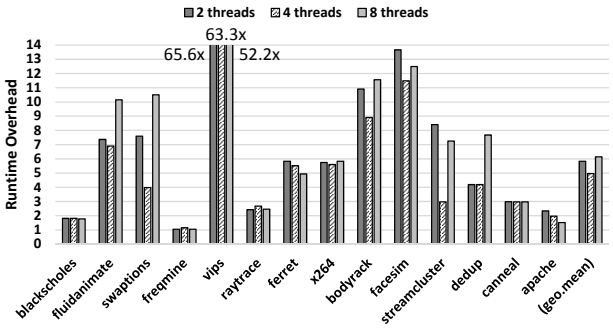


Figure 3.8: Scalability of TxRace

source state-of-practice happens-before based data race detector. For each memory location and synchronization variable, TSan keeps track of happens-before order information into shadow memory. TSan supports compile-time instrumentation for data race detection using Clang frontend [195] and LLVM passes [115]. For simplicity, we instrument fast/slow path codes together into the original program. For example, the same memory access hook is instrumented for both fast/slow paths. Depending on the fast or slow path, the hook performs TSan data race detection for slow path or it does nothing for fast path. For better performance, it would be ideal to clone the codes and have separate fast/slow path codes to remove the redundancy similar to [113, 191]. We leave this optimization as a future work.

application	committed transactions	conflict aborts	capacity aborts	unknown aborts	TSan races	TxRace races	original time(ms)	TSan time(ms)	TxRace time(ms)	TSan overhead	TxRace overhead
blackscholes	131105	2	0	7	0	0	253	467	460	1.85x	1.82x
fluidanimate	17778944	696789	10321	36614	1	1	539	8217	3724	15.23x	6.9x
swaptions	160640076	2599	557497	54317	0	0	868	5875	3446	6.77x	3.97x
freqmine	84	0	3	26	0	0	3973	55611	4569	14x	1.15x
vips	707547	16793	23403	14985	112	79(*)	953	1139087	60320	1195x	63.28x
raytrace	143	12	0	14	2	2	4546	23130	12203	5.09x	2.68x
ferret	208052	379	2413	4263	1	1	1060	11390	5852	10.74x	5.52x
x264	36808	245	423	5358	64	64	595	3837	3332	6.45x	5.6x
bodytrack	9950991	36004	47050	2004723	8	6(*)	503	6429	4479	12.78x	8.9x
facesim	12827334	1611	3372	38563	9	8(*)	2439	89242	28027	36.59x	11.49x
streamcluster	756908	170805	230	832	4	4	1430	39042	4253	25.9x	2.97x
dedup	2185219	106618	13889	40177	0	0	2748	13292	11513	4.84x	4.19x
canneal	3200570	25187	2896	106419	1	1	3499	15367	10375	4.39x	2.97x
apache	310781	227	446	9793	0	0	6916	21089	13600	3.05x	1.97x
geo.mean										11.68x	4.65x

Table 3.1: TxRace Execution Statistics and Performance.

3.8 Evaluation

Our evaluation answers the following questions:

- What is the overhead of TxRace data race detection? Is it efficient?
- Does TxRace effectively detect data races? How many false negatives are there?
- Is TxRace cost-effective compared to other approaches? Is it better than a sampling-based approach, or a full happens-before based detector?

3.8.1 Methodology

We ran experiments on a 3.6GHz quad-core Intel Core™i7-4790 processor, with 16GB of RAM, running Gentoo Linux (kernel 4.0.4). Intel's Restricted Transactional Memory (RTM) is used for HTM-based fast path data race detection. Intel Haswell processor supports the same number of concurrent transactions as the hardware threads available, which is four (eight with hyperthreading) in our case. On the other hand, Google's ThreadSanitizer (TSan) is used for software-based slow path data race detection.

TxRace was evaluated using 1) PARSEC benchmark suite [41] that is designed to be representative of next-generation shared-memory programs including emerging workloads; and 2) Apache web server [1]. We used *simlarge* input for all the 13 application in PARSEC, and tested Apache using ab (ApacheBench) by sending 300,000 requests from 20 concurrent clients over a local network. Performance was reported in terms of overhead with respect to the original execution time without data race detection. We compare our system (named *TxRace* in the result) with off-the-shelf Google's ThreadSanitizer (named *TSan*). All results are the mean of five trials with four worker threads (except the scalability analysis).

3.8.2 Performance Overhead

Table 3.1 shows the TxRace execution statistic, the number of detected data races, and overall performance results. The first column provides the application name. The next four columns show transaction statistics during HTM-based fast path data race detection: the number of total committed transactions, the number of data conflict aborts, the number of capacity (overflow) aborts, and the number of unknown (unspecified) aborts. The next two columns give the number of races detected by TSan and TxRace. The applications in which TxRace cannot detect all the races reported by TSan are marked with asterisk. The next three columns show the original, TSan, and TxRace execution times. The first is the execution time of the original application (with no transactions, memory hooks, etc.); the second is the execution time when TSan is used; the third is the execution time of our system TxRace. The last two columns show TSan's and TxRace's overhead with respect to the original execution.

Examining the results, we see that TxRace's overhead is generally low. On average, TxRace reduces runtime overhead of dynamic data race detection from 11.68x to 4.65x (geometric means), showing 60% reduction ratio. For some applications such as *vips* and *streamcluster*, TxRace achieved more than 10 times speedup over TSan.

Figure 3.7 shows a breakdown of the overhead normalized to the original execution time (baseline) for all benchmarks. The black portion in each bar (xbegin/xend) represents the pure fast path overhead in which transactions are executed, but no slow path is taken even when they get aborted (simply run untransactionalized code). For most applications, this overhead is pretty low (the geometric mean of 17%), except *swaptions* and *streamcluster*. Upon further investigation, we found out that these two applications have tight loops that have system calls in the loop body. In this case, TxRace ends and begins new transac-

tions around the system calls. This results in tight short transactions whose management cost now becomes dominant. The next overhead comes from handling aborts due to data conflicts (157%), which includes running slow path software-based data race detection. This low-overhead result shows the benefits of using HTM-based potential data race detection beforehand, which allows TxRace to selectively perform the software-based dynamic data race detection only for small fraction of execution intervals. Finally, the remaining performance overhead comes from handling capacity and unknown aborts (126% and 66%, respectively). To achieve small false negatives, TxRace takes the conservative approach of using slow path software-based detector to monitor program regions that fast path HTM cannot cover. We envision that if there is an ideal HTM such that a transaction aborts only if there is a data conflict and do not have capacity/unknown aborts, then the runtime overhead of TxRace would be improved significantly as TxRace only falls back to slow path when necessary (only when a data conflict occurs).

Figure 3.8 shows the scalability of TxRace. We varied the number of worker threads from 2, 4 to 8, and measured the runtime overhead normalized to the original execution time with 2, 4, and 8 worker threads, respectively. For comparison of two and four thread cases, some applications such as *swaptions* and *streamcluster* show lower runtime overhead for four thread cases, but most of the remaining applications show small differences in normalized runtime overhead. Upon further investigation, we found out that the number of capacity aborts (geometric mean of 644 vs. 474) and unknown aborts (geometric mean of 8377 vs. 4651) decreases from two to four thread cases. The reduction in capacity aborts makes sense as many applications in PARSEC benchmark take advantage of data parallelism, and thus each worker thread is likely to have smaller dataset with more worker threads. On the other hand, the number of conflict aborts (geometric mean of 244 vs. 1242) increases from two to four thread cases (likely due to increased concurrency). After all, the mixture of the increase

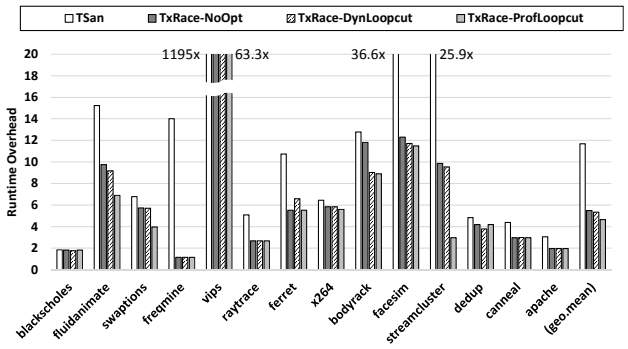


Figure 3.9: Effectiveness of loop-cut optimization

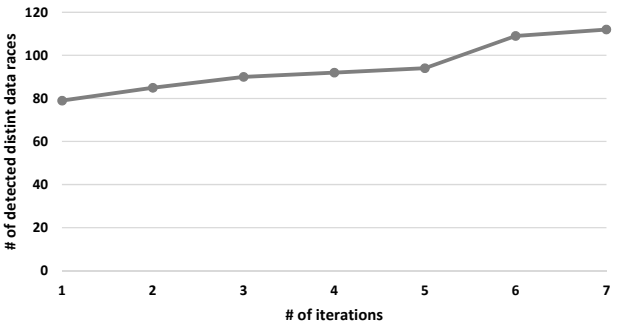


Figure 3.10: The number of detected distinct data races across multiple runs for *vips*

and the decrease in transactional aborts causes different applications to show variation in performance overhead.

Another interesting result of this experiment is the high overhead incurred for some applications with eight threads (e.g., *fluidanimate*, *swaptions*, *streamcluster*, and *dedup*). Examining the results, we found that the number of unknown aborts increases significantly for eight thread case (geometric mean of 42251, which is 5x and 9x more than two and four thread cases, respectively). As Intel Haswell processor does not provide additional information regarding unknown aborts, further investigation was not possible, but we suspect that eight concurrent transactions enabled by hyperthreading might lead to increased unkonwn aborts.

Finally, we evaluate the effectiveness of the loop-cut optimization discussed in the Section 3.4.3. Figure 3.9 presents the normalized runtime overhead of TSan and three different types of TxRace. They differ from each other based on how they handle a transaction that includes a loop with a large number of iterations, causing capacity aborts frequently. *TxRace-NoOpt* stands for the basic scheme without optimization that TxRace simply falls back to slow path every time when a transaction gets aborted for the capacity reason. *TxRace-DynLoopcut* represents the optimized scheme that for a transaction including a loop, TxRace dynamically learns the loop iteration count (called *loop-cut-threshold*) that do not

cause a capacity abort at runtime. When the transaction gets aborted, TxRace falls back to the slow path at first. However, when the same loop is executed next time, TxRace uses the loop-cut-threshold to terminate the transaction early in the middle of loop iterations and starts a new transaction to avoid capacity aborts. *TxRace-ProfLoopcut* is similar to the above dynamic scheme, but it profiles the program with representative input to collect the initial loop-cut-threshold, and avoids even the very first capacity aborts. Figure 3.9 shows the benefits of leveraging the loop-cut optimization. In all cases, TxRace is more efficient than TSan. On average, *TxRace-ProfLoopcut* shows the best result (4.65x) in terms of performance overhead, and *TxRace-DynLoopcut* which does not require profiling the threshold also performs better than TSan (5.34x).

3.8.3 False Negatives

In this section, we study false negatives of TxRace. HTMs detect data conflicts between transactions that are concurrently overlapped in time. As a result, similar to other overlap-based data race detectors [43, 56, 75, 78], TxRace may miss data races if they happen far apart in time. The sixth and seventh columns of Table 3.1 represent the average number of data races reported by happens-before based TSan and our overlap based TxRace. Here, each race is in a form of racy instruction pair, and we count the number of static instances.

There are three applications (*vips*, *bodytrack*, and *facesim*) that TxRace detects less data races than TSan. It turns out that the missed three cases of *bodytrack* and *facesim* are due to the common *initialization* idiom, in which a data structure is allocated within a thread and initialized without any synchronization while the structure is still local to the thread, and then it becomes accessible to other threads, by adding it to a global data structure. For example in *facesim*, a structure is initialized when a thread pool is created at the beginning

of program execution, then it becomes shared at a later time. TxRace missed such races because conflicting accesses do not overlap.

On the other hand, for *vips*, though the number of data race found for each test run remains about the same (average of 79), we observed that TxRace actually finds different sets of data races across different runs. This makes sense because TxRace's nature of the overlap-based detection makes it sensitive to underlying OS scheduler. Figure 3.10 shows that when we accumulate the distinct data races detected, TxRace can find all the data races (112) found by TSan after seven runs. Note that for *vips*, TxRace (63.3x) is order of magnitude faster than TSan (1195x).

3.8.4 Cost-Effectiveness of Data Race Detection

TxRace is complete (no false positive) but unsound (some false negative). In essence, TxRace aims to be a cost-effective solution that exploits a critical tradeoff of soundness for performance. To quantitatively evaluate how cost-effective TxRace is, we rely on a popular economic analysis term called *cost-effectiveness ratio* where the denominator is the effectiveness and the numerator is the cost. The original ratio is inverted and redefined for the context of data race detection to quantify how cost-effective it is as follows:

$$CostEffectiveness = \frac{Race\_Detection\_Effectiveness}{Race\_Detection\_Cost}$$

As a metric to evaluate the data race detection effectiveness, we use *recall* that is commonly used to measure the quality of classifiers in information retrieval and bug detection communities [42, 110, 122, 213]. Intuitively, high recall leads to less false negatives (undetected

application	overhead	recall	cost-effectiveness
blackscholes	0.99	1	1.02
fluidanimate	0.45	1	2.21
swaptions	0.59	1	1.7
freqmine	0.08	1	12.17
vips	0.05	0.71	13.32
raytrace	0.53	1	1.9
ferret	0.51	1	1.95
x264	0.87	1	1.15
bodytrack	0.7	0.75	1.08
facesim	0.31	0.89	2.83
streamcluster	0.11	1	8.71
dedup	0.87	1	1.15
canneal	0.68	1	1.48
apache	0.65	1	1.55
geo.mean	0.38	0.95	2.38

Table 3.2: Cost-Effectiveness of TxRace vs. TSan

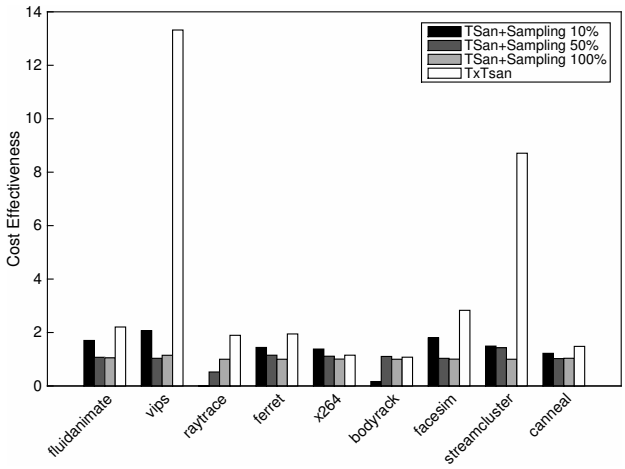


Figure 3.11: Cost-Effectiveness of TxRace vs. Sampling

data races). In the context of data race detection, *recall* is defined as follows:

$$recall = \frac{|Reported\_Data\_Races \cap Real\_Data\_Races|}{|Real\_Data\_Races|}$$

For comparison to TSan, *Real_Data_Races* is defined as the data races reported by TSan. To calculate the cost effectiveness (CE), we use TxRace's runtime overhead normalized to TSan's. Table 3.2 summarizes how much more cost-effective TxRace is compared to TSan for each benchmark application (here TSan's CE is 1). TxRace turns out to be 2.38x (geometric mean) more cost-effective than TSan across the benchmark applications. This is mainly because in TxRace, only small portion of memory accesses are investigated for software-based data race detection (slow path). On the other hand, the majority of memory accesses are dealt with by transactional execution (fast path) at a very low runtime cost.

To justify such a high cost-effectiveness of TxRace, we also compare TxRace with TSan with sampling. Sampling memory operations is an intuitive way to reduce runtime overhead of dynamic data race detection. However, it also comes with false negative issues because some data races might be missed at a low sampling rate. To study if TxRace is more cost-effective

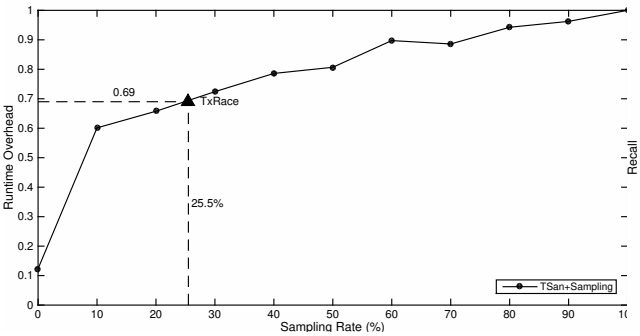


Figure 3.12: Runtime overhead for *bodytrack*

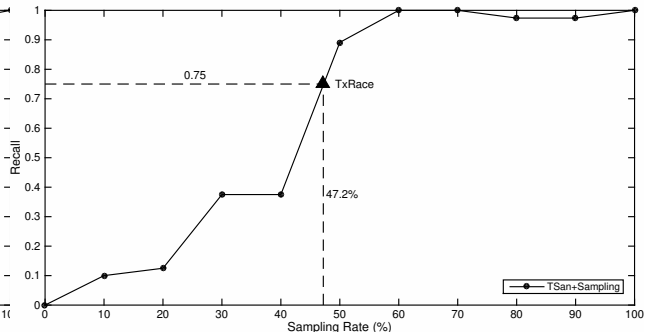


Figure 3.13: Recall for *bodytrack*

than sampling, we vary the sampling rate and measure the resulting runtime overhead and the recall of TSan for every benchmark. As a represetative application, we present the result of *bodytrack* in detail. Figure 3.12 shows the runtime overhead normalized to 100% sampling (full coverage), and Figure 3.13 shows the recall at different sampling rates while treating 100% as an oracle. As expected, both the runtime overhead and recall increase as the sampling rate increases. On the other hand, the normalized runtime overhead and the recall of TxRace are 0.69 and 0.75, respectively. This implies that TxRace adds an overhead equivalent to sampling about 25.5% of memory operations, but its recall is equivalent to 47.2% sampling, which shows its cost effectiveness.

Lastly, Figure 3.11 presents the cost-effectiveness of TxRace compared to TSan with sampling across nine applications in which TxRace/TSan detect at least one data race. For some applications such as *fluidanimate*, *vips*, and *facesim*, 10% sampling turns out to be more efficient than 50% or 100% sampling cases. It turns out that the data race in such applications manifest often at runtime, thus they get detected even at the low sampling frequency. In other words, the number of dynamic instances of the data race is quite high even though the number of static instance (unique race condition) is small (e.g., one for *fluidanimate*), thus reaching the recall of almost 1 (no false negative) at the low frequency. After all, for almost all applications execpt *x264*, TxRace outperforms TSan with sampling in terms of

the cost-effectiveness.

3.9 Summary

The spread of shared-memory multiprocessor architectures has spurred development of multithreaded programs. However, such programs are subject to concurrency bugs including data races. Unfortunately, traditional dynamic data race detectors are too slow to use in many cases. In this chapter, we improve the run-time overhead of dynamic data race bug detectors (**PS1**). We presents TxRace, a new software dynamic data race detector that exploits commodity hardware transactional memory support to enable dynamic data race detection with a low run-time overhead (**TS1**). Leveraging existing HTM support allows TxRace to use a precise but expensive dynamic data race detector only for a small fraction of the whole execution in an on-demand manner, leading to performance improvement. The experiment results show that TxRace achieves run-time overhead reduction of dynamic data race detection by 60% on average with only a few false negatives (high recall of 0.95). TxRace is published at [230].

Chapter 4

Practical Data Race Detection for Production Use

In previous chapter, we presented a dynamic data race detector leveraging commodity hardware transactional memory, which have 4.65x run-time overhead. However, due to the undeterministic behaviour of data races it is often hard to find them all during testing and they often manifest themselves in a production environment. Therefore, people are interested deploying dynamic data race detectors in production. In this chapter, we presents ProRace, a dynamic data race detector practical for production runs. It is lightweight, but still offers high race detection capability. To track memory accesses, ProRace leverages instruction sampling using the performance monitoring unit (PMU) in commodity processors. Our PMU driver enables ProRace to sample more memory accesses at a lower cost compared to the state-of-the-art Linux driver. Moreover, ProRace uses PMU-provided execution contexts including register states and program path, and reconstructs unsampled memory accesses offline. This technique allows ProRace to overcome inherent limitations of sampling and improve the detection coverage by performing data race detection on the trace with not only sampled but also reconstructed memory accesses. Experiments using racy production software including *apache* and *mysql* shows that, with a reasonable offline cost, ProRace incurs only 2.6% overhead at runtime with 27.5% detection probability with a sampling period of 10,000.

4.1 Introduction

Despite of extensive in-house testing races often exist in deployed software and manifest in customer usage [111, 180, 184]. These test escapes occur because data races are highly sensitive to thread interleavings, program inputs, and other execution environments that testing cannot completely cover [43]. For the same reasons, data races are notoriously difficult to reproduce and fix after being observed in a production run. Consequently, there is an urgent need for a lightweight data race detector that can monitor production runs.

In production settings, it makes sense to trade off soundness (may miss data races) for performance. Sampling [47, 78, 135, 184] has been proposed as a promising technique to address the problem. However, LiteRace [135] and Pacer [47] still incur unaffordable slowdown for some applications (e.g., Pacer [47] adds 86% overhead at the 3% sampling ratio) due to code instrumentation based runtime checks. Though DataCollider [78] uses hardware breakpoint support instead, their detection coverages are limited to sampled accesses only. RaceZ [184] pioneered the use of hardware performance monitoring unit (PMU) to sample memory accesses, but it has to keep the low sampling frequency for performance thereby compromising the detection coverage.

LiteRace [135] and Pacer [47] pioneered the use of sampling for reducing the overhead of dynamic data race detection. LiteRace focuses on sampling more accesses in infrequently-exercised code regions, based on the heuristic that for a well-tested application, data races are likely to occur in such a cold region. On the other hand, Pacer uses random sampling and thus its coverage is approximately proportional to the sampling rate used. However, these code instrumentation-based race detectors cause an unaffordable slowdown for some applications, and their detection coverage is limited to the sampled accesses only. For example, though LiteRace shows low 2-4% overhead for Apache, it makes CPU-intensive applications 2.1-2.4x

slower, and incurs 1.47x slowdown on average for their tested applications. Similarly, Pacer also reports the average of 1.86x overhead at the 3% sampling frequency.

DataCollider [78] and RaceZ [184] avoid code instrumentation and thus incur a very low overhead, but suffer from low detection coverage. DataCollider [78] makes use of hardware debug breakpoints. After sampling a code/memory location, it sets a data breakpoint and inserts a time delay. A trap during this delay indicates a conflicting access from another thread. Though longer timing delays increase the likelihood of overlapping data races, they also increase the overhead. In addition, hardware restrictions limit the number of concurrently monitored memory locations to four in the latest x86 hardware [101].

RaceZ leverages Intel's PEBS to sample memory accesses. However, due to its reliance on the inefficient Linux PEBS driver, RaceZ has to use a low sampling frequency for performance, thereby compromising the detection coverage. RaceZ also attempts to reconstruct unsampled memory accesses, but its scope is limited to a single basic block. This work shows that ProRace has much less overhead, but detects significantly more data races compared to RaceZ.

Another line of work takes a hybrid static-dynamic approach. RaceMob [111], a recent low-overhead solution, employs static analysis [204] to compute potential data races, and crowdsources runtime race checks across thousands of users. To limit the overhead each user may experience, RaceMob requires a large number of runs to distribute checks, and the number of runs required depends on the precision of the static analysis. Elmas et al. [76] and Choi et al. [58] are other examples that make use of static data race analysis to reduce runtime cost. In spite of its benefits, static analysis often suffers from precision and scalability issues for large-scale applications, and the recompilation requirement is often not a viable option in production settings.

We design ProRace (Chapter 4), a new practical sampling-based data race detector for production runs. It is *lightweight*, minimally affecting the application execution; *transparent*, requiring neither recompilation nor static analysis; and *effective*, ensuring high race detection coverage.

It consists of online program tracing and offline trace-based data race analysis. Though offline analysis is required, the principal advantage of It is that very low runtime overhead of the online part enables It to monitor real-time, interactive, or internetworked applications at nearly full speed.

It makes use of the hardware PMU in commodity processors to monitor an unmodified program at a very low overhead. To be specific, It samples memory accesses using Intel's Precise Event Based Sampling (PEBS) [99]. It's newly designed PEBS driver avoids unnecessary kernel-to-user copying and sampled data processing, reducing overhead by more than half compared to the latest Linux PEBS driver. This allows It to take much more samples for a given performance budget, enhancing its detection coverage.

During the offline phase, It reconstructs unsampled memory accesses to overcome the inherent limitation of sampling and to increase data race detection coverage further. The key idea is to replay the program from each sample and reconstruct the addresses of other memory instructions. Over the sampling, PEBS provides not only the sampled instruction but also its architectural execution context (e.g., register states) at sample time. It re-executes the program binary starting from each sampled instruction with the register states, and re-calculates the addresses of unsampled memory operations while emulating register and memory states.

Furthermore, to recover more memory accesses around each sample, It collects the complete control-flow trace using Intel's Processor Trace (PT) [100], a new feature in the Intel proces-

sor's PMU, at runtime. The control-flow information guides which path to take during the offline replay, enabling It to reproduce many other unsampled memory operations preceding and following each sample along the observed program path.

Finally, It analyzes the recovered memory trace and the synchronization trace, to detect data races using the happens-before based race detection algorithm [79].

This work makes the follwing contributions:

• ProRace presents a lightweight, transparent, and effective data race detector that can be easily deployed to monitor production runs.

• ProRace proposes a new methodology to reconstruct unsampled memory addresses using the control-flow trace collected at runtime. To the best of our knowledge, ProRace is the first software scheme that demonstrates how commodity hardware support for control-flow tracing can be used to enable the forward and backward reconstruction of unsampled memory trace. The proposed solution can benefit future research on runtime monitoring beyond race detection.

• This paper describes a PEBS driver that is many more efficient than the state-of-the-art Linux PEBS driver.

• The experiments using production software including *apache* and *mysql* show that ProRace can detect significantly more races than RaceZ, a PEBS based race detector, at a much lower overhead.

4.2 Overview

The goal of ProRace is to provide lightweight yet effective race detection for practical use in a production environment. We envision a production environment similar to Google/Facebook's real-world datacenter in which various traces of production applications are already collected for monitoring purposes, and dedicated analysis machines exist in the datacenter to process the collected trace [110, 168]. In such environment, runtime monitoring overhead is much more critical concerns than the size of trace and offline analysis overhead. Production and analysis machines share a separate network, and thus writing a trace has a minimal impact on the QoS of production applications that use another network. Analysis machines can periodically process the trace to delete the ones analyzed in prior periods.

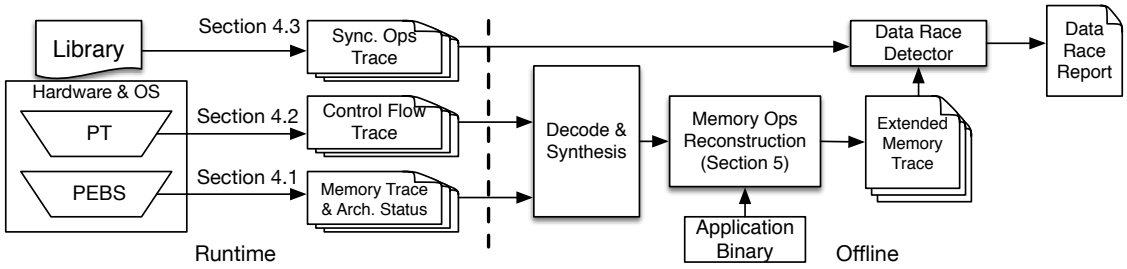


Figure 4.1: Overview of the ProRace Architecture.

Figure 4.1 shows an overview of ProRace's two-phase architecture: online program tracing and offline data race detection. The online stage leverages the hardware PMU to trace a program execution at low overhead. Specifically, PEBS is used to collect the sampled memory access trace. PEBS provides both the sampled instruction and the architectural execution context (e.g., register states) at the sample time. PT is used to obtain the complete control-flow trace. The online stage also tracks the synchronization operations for later use in data race detection.

The offline stage first combines the memory access and control flow traces into a time-

synchronized trace. Next it reconstructs *unsampled* memory operations. This is the critical step that allows ProRace to achieve higher detection coverage than other sampling-based approaches. Using the sampled instruction, register states, and control-flow information, ProRace replays the program and recomputes the addresses of unsampled memory accesses around each sample. The unsampled memory instructions whose target addresses can be reconstructed during this step are included in an *extended* memory access trace. Combining this with the synchronization trace, ProRace performs happens-before based data race detection using the FastTrack [79] algorithm to detect data races.

ProRace improves existing PMU-sampling-based data race detection in three ways. First, ProRace presents a PEBS driver much more efficient than the latest Linux PEBS driver. The improved design allows ProRace to take more samples for a given performance budget, enhancing its race detection coverage. Second, ProRace recovers unsampled memory accesses. ProRace re-executes the program binary starting from each sampled instruction with the PEBS-provided register states reconstructing the unsampled memory accesses while emulating register and memory states. Third, ProRace uses the PT-collected control-flow trace to choose which path to take during the offline binary re-execution. This permits ProRace to recover many other unsampled memory operations around each sample along the observed program path.

4.3 Lightweight Program Tracing

This section presents how ProRace traces a program execution at low overhead. At runtime, ProRace collects three type of traces: memory access samples, control-flows, and synchronization operations.

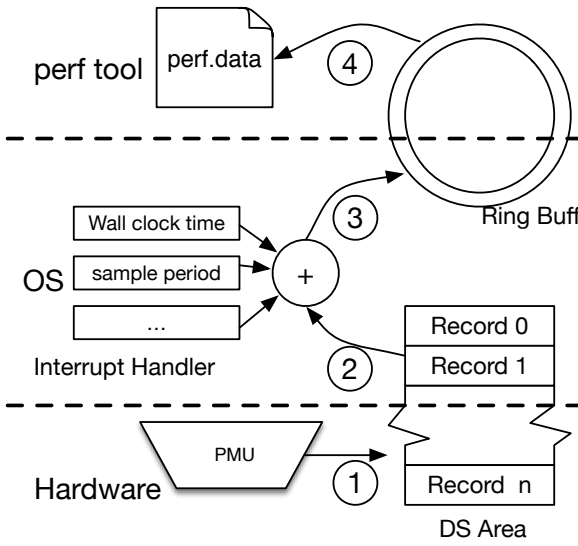


Figure 4.2: The vanilla Linux PEBS driver

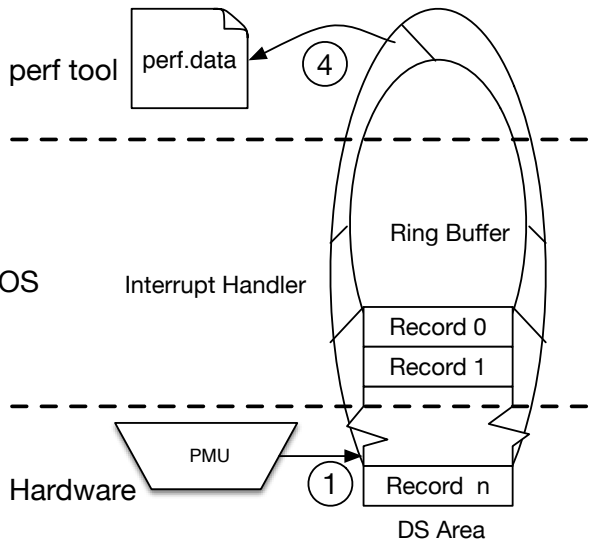


Figure 4.3: The ProRace PEBS driver

4.3.1 PEBS-based Memory Access Sampling

ProRace samples memory accesses using PEBS [99]. PEBS users can specify types of events to monitor such as retired memory instructions and taken branches, as well as whether to sample user-level or both user- and kernel-level events. ProRace tracks only the user-level retired load and store instruction events because of its interests in application memory accesses for data race detection.

PEBS enables users to set a *sampling period* k for each monitored event type. After every k events of a given type, PEBS delivers the sampled event along with its architectural execution context at the sample time such as register values, the time stamp counter (TSC)[1], but not memory states, to the corresponding listener.

Care must be taken when choosing the sampling period. Small values of k yield more samples but higher performance overhead. In addition, samples may be dropped if the kernel finds

[1]In old Intel processors, the PEBS samples did not include the time stamp, and the OS interrupt handler logged its wall-clock time during the processing. As a result, there was a small timing gap between the actual hardware sample time and the interrupt handler logging time. However, this is no longer an issue in recent processors such as Skylake and Broadwell.

that too much time has been spent on the interrupt handling.

The Current Linux PEBS Driver

While the previous version of the Linux PEBS driver delivered every event using an overflow interrupt, a mechanism called *Debug Store (DS)* was added in the 4.2 Linux kernel to reduce the interrupt frequency. Figure 4.2 illustrates the interactions between the hardware PEBS, the OS interrupt handler, and the user-level *perf* tool.

DS permits PEBS to automatically store samples in a kernel-space buffer referred to as the *DS save area* whose default size is 64 KB (step ①). The interrupt is delivered only when the DS buffer is nearly full, reducing the frequency of interrupts.

On each interrupt, the OS interrupt handler processes the raw 'PEBS events', adding additional information such as wall-clock time, sample size, and sample period (step ②), and yielding 'perf events'. It then copies the perf events into another buffer, a ring-buffer shared with the user-land *perf* tool, resetting the DS save area for further PEBS events (step ③).

Finally, the *perf* tool polling on the ring-buffer commits the perf events to a file (step ④). Since the user-land perf tool may be configured to monitor incoming data from different cores, and store them into the same file, the events in the file may not be ordered sequentially. Thus, it reads the entire file later before its exit to sort all events and include other information.

Though DS support reduces the runtime overhead in using PEBS compared to the naive interrupt-per-sample mechanism, our experimental results show that a sampling period more frequent than 10K-100K will still incur slowdowns approaching 10%.

ProRace's New PEBS Driver

ProRace presents a new PEBS driver that significantly lowers the performance overhead in using PEBS. The new design makes it possible to collect more samples for the same performance cost. Figure 4.3 shows our new design incorporating the following changes:

First, ProRace eliminates expensive kernel-to-user copying by maintaining a single ring buffer named *aux-buffer*. The ring buffer is partitioned into multiple 64 KB segments. Initially, ProRace provides PEBS with one segment of the ring buffer; when PEBS finds it full and raises an interrupt, the OS interrupt handler simply proffers the aux-buffer's next available segment. The user-level *perf* tool eventually comes into play, dumping the segment filled with records into the file and making it available for further tracing. In this design, the interrupt handler need only swap the segment locations for PEBS similar to conventional double-buffering. The Linux driver for (newer) Intel's PT incorporates a similar single buffer design, but it is not used in the PEBS driver.

Second, ProRace skips data processing irrelevant to data race detection during PEBS sample handling. Specifically, ProRace does not add the metadata information (step ② in Figure 4.2).

Lastly, given a sampling period P, the sampling period is initially set to a random value between one and P. At the first event the sampling period is changed to P. This enables ProRace to start sampling at a random location per thread on each run, increasing its sampling diversity to ultimately improve its race detection capability.

Experimental results in Section 4.6.2 show that the new driver reduces runtime overhead significantly, making it possible for applications to use a small sampling period.

4.3.2 PT-based Control-flow Tracing

ProRace uses Intel's PT [100] to collect program control flows. PT is an extension to the PMU architecture for Intel's Broadwell and Skylake processors. At runtime, PT records the executed control-flow of the program in a highly-compressed format. Unlike event-based PEBS, PT keeps track of complete control-flow information including (indirect) branch target and call/return information without loss of precision. Nonetheless, PT incurs only a very small overhead because the tracking is done off the critical path and by hardware. This is significant improvement over previous (relatively) high overhead and limited tracking features such as Branch Trace Store (BTS) and Last Branch Record (LBR) in old processors.

ProRace's PT driver also implements the code-region based control-flow tracing feature. The PT hardware allows users to specify four memory regions of interest from which to collect the program control-flow. ProRace is configured to monitor only main executable memory regions because of its interests in detecting application data races (assuming no Just-In-Time compilation). Depending on use cases, dynamic library code regions may be included, or static library code regions may be excluded, by examining the symbol table.

The memory access trace collected by PEBS and the control flow trace collected by PT can be easily combined for offline processing because both traces include the per-core TSC value.

4.3.3 Synchronization Tracing

ProRace uses happens-before based data race detection [79] for precision (no false positives), but offloads the expensive vector-clock processing to the offline phase. At runtime, ProRace collects per-thread synchronization logs along with its type (e.g, lock/unlock), variable (e.g., lock variable address), and TSC value. The per-thread logs can be easily synchronized offline because recent processors support invariant TSC [86] that is synchronized among cores and

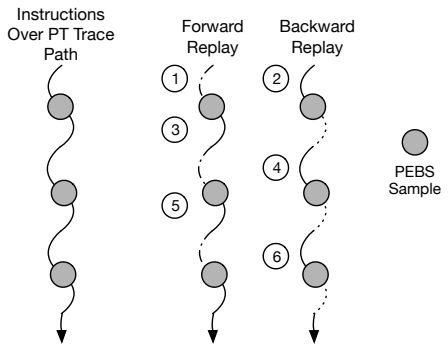


Figure 4.4: Forward and Backward Replays.

runs at a constant rate.

For transparency, ProRace uses *LD_PRELOAD* to redirect standard *pthread* functions to ProRace instrumented functions. In addition, ProRace tracks dynamic memory allocation/deallocation. Suppose that one object is freed, and another object happens to be allocated to the same memory location. There can be no race condition between two different objects, but a data race detector may falsely report one as their memory addresses are the same. To avoid this kind of false positive, many data race detection tools keep track of malloc and free, and so does ProRace.

4.4 Recovering Unsampled Memory Accesses

ProRace leverages PMU-based instruction sampling to collect memory accesses. As with all the sampling-based race detectors, it might end up with false negatives due to unsampled memory accesses. To overcome the inherent limitation of sampling, ProRace reconstructs unsampled memory accesses offline by re-executing the program binary around each PEBS-sampled instruction with forward replay (Section 4.4.1) and backward replay (Section 4.4.2). In addition, ProRace leverages full control-flow information recorded by PT to guide which path to execute during both replays.

For each PEBS sample, ProRace alternates forward and backward replays following the observed program path as shown in Figure 4.4. Basically, the forward replay corresponds to the re-execution of the unsampled instructions between the current and the next samples, while the backward replay to that of those preceding the current sample for dealing with the instructions missed by the forward replay. ProRace repeats the replays until there is no more PEBS sample to be processed. The rest of this section details the path-guided binary re-execution and how it can reconstruct unsampled memory accesses.

4.4.1 Forward Replay

When an event is sampled, PEBS not only offers precise instruction location of the event, but also provides the architectural states such as the entire register file contents at the sample time. By leveraging such execution contexts as inputs, ProRace re-executes the program binary from each PEBS sample point over the program path reconstructing the addresses of the memory operations. Such path-guided binary re-execution is called *forward replay*.

For each PEBS-sampled instruction, ProRace restores the register file contents, and attempts to execute every following instruction over the program path until the next PEBS-sample point is reached. For each instruction being executed, ProRace checks if the operands are available at the time of the instruction execution. For this purpose, ProRace keeps track of the architectural status by bookkeeping all the register and memory values in a special hash table called *program map*.

ProRace simply treats every memory location as *unavailable* in the first place. The destination register of load instructions becomes unavailable when they read from unavailable memory locations. If all the operands of an instruction being replayed are not *available*, ProRace simply skips the instruction setting all its outputs as unavailable. Otherwise,

ProRace executes the instruction updating the resultant architectural status such as registers and memory locations in the program map. Note that the memory emulation requires a special care for correctness, and thus it is used in a limited fashion. By default, when any available register is written to a certain memory location, ProRace bookmarks the value for a later access during the replay in the program map and treats the location as *available*. However, when ProRace hits a system call or an unavailable instruction, it conservatively invalidates emulated memory states. Moreover, the memory emulation might lead to incorrect memory address reconstruction after the racy access (i.e., conflicting write) from other threads. To address this problem, when a race is detected on the emulated memory location in a later phase, ProRace invalidates the memory location and regenerates the trace from that racy point (i.e., conflicting read) with the unavailable register value. Thus, ProRace is safe as it never uses racy memory location during the trace regeneration.

While the forward replay progresses further, more registers become unavailable by the load instructions reading from unavailable memory locations. Thus, at some point, ProRace may end up with a situation where no register is available. One might think that the forward replay cannot proceed anymore because no more instruction can be executed due to the lack of available operands. However, continuing the replay even across the point where all registers become unavailable can capture some unsampled memory accesses that would otherwise be impossible to reconstruct. For example, if memory instructions leverage PC-relative instructions, e.g., `mov offset(%rip)` in x86-64, ProRace can figure out the memory location by adding the offset to `%rip` which is always available as an instruction pointer (PC). By taking advantage of the full control-flow trace recorded by PT, ProRace performs the forward replay across basic block boundaries until it reaches the very next PEBS-sampled instruction.

Figure 4.5 shows how ProRace reconstructs many unsampled memory accesses using forward

```
0: mov    %rax,0x18(%rsp)
1: movslq 0x0(%rbp,%rbx,4),%rdx
2: mov    (%r15,%rbx,8),%rsi
3: mov    0x8(%rsi),%rax
4: mov    %r10,%rdi
5: mov    0x8(%r14),%rax
6: add    %rax,%r13
7: xor    %eax,%eax
8: mov    %r13,0x8(%r14)
9: mov    0x18(%rsp),%rcx
10: mov    (%r15,%r12,8), %rsi
```

Figure 4.5: Example for Forward and Backward Replay

replay with a real-world example extracted from *Apache*. Suppose ProRace sampled the `mov` at a line 0 and recorded the register states at the sample time. After restoring all the register values, ProRace performs the forward replay for the following instructions. Here, the forward replay can successfully reconstruct the memory addresses of the instructions at line 1, 2, 5, 8, 9 and 10 since their registers used for the address calculation are all available.

However, the memory address of the instruction at line 3, i.e., `mov 0x8(%rsi),%rax`, cannot be reconstructed because `%rsi` reads from memory location that is currently unavailable by the instruction at line 2, i.e., `mov (%r15,%rbx,8),%rsi`. To solve this problem, ProRace performs the backward replay right after the forward replay.

4.4.2 Backward Replay

Forward replay cannot reconstruct the address of memory operations if the register operand of memory instructions is unavailable, or if the address is not obtained by PC-relative addressing. This motivates ProRace to leverage two forms of backward replay to reconstruct the memory addresses skipped by the forward replay: backward propagation and reverse execution.

Backward Propagation

The key observations is that many of unavailable registers can be recovered by consulting the next PEBS-provided execution contexts where all the register values are available. More precisely, the backward replay can reconstruct the memory access whose register operand became unavailable during the forward replay, provided the register has not been updated before the next PEBS-sampled instruction. Fortunately, according to empirical results, the registers used for memory address calculation often have a long live-range [140] after they become unavailable during the forward replay.

In light of this, ProRace back-propagates all the register values restored at the very next PEBS sample to the instructions where each register has been most recently updated. For this purpose, the forward replay marks such instructions checkpointing the register file at the time the register is updated. In addition, the forward replay keeps track of the youngest one among the instructions as an entry point of the later backward replay. Once all the register back-propagation is performed, ProRace simply jumps to the youngest instruction and resumes the re-execution there. In a sense, the backward replay can be considered as yet-another forward replay starting from a different location, i.e., the youngest instruction, not the current PEBS-sampled instruction.

Figure 4.5 also shows how the backward replay reconstructs an unsampled memory access that the forward replay cannot deal with. Suppose ProRace sampled the instruction at line 10. This allows ProRace's backward analysis to restore the value of `%rsi`, which is not possible for the forward replay to deal with. In this way, ProRace can successfully reconstruct the memory address of the instruction at line 3 using the restored register.

Reverse Execution

The second type of ProRace's backward replay is based on *reverse execution* [52, 63, 125]. In its simplest form such as register-to-register copy, the reverse execution can restore both register values based on the equality as long as at least one of them is known. It is also possible to restore the register used as an operand of arithmetic instructions provided the other operand (register) is known during the backward replay. For example, the reverse execution can restore the `%rdx` operand of an instruction (`%rax = %rdx + $offset`), if the other operand (`%rax`) is already available by subtracting the `$offset` from `%rax`. ProRace's backward replay engine currently supports reverse execution of integer arithmetic instructions such as additions and subtractions.

Note that once an unavailable register is restored by the reverse execution, ProRace can restore others that have a dependence on that register. As PT provides the program path, ProRace only needs to track the data dependencies, and triggers forward and backward replays iteratively until they reach the fixed point [140] where no further restoration is found. This simple yet effective technique allows the backward replay to go backward further possibly reconstructing more unsampled memory accesses.

4.5 Implementation

The online tools for ProRace consists of two parts: kernel-level PMU drivers and user-land *perf* tool. The new PEBS driver is implemented based on the Linux kernel version 4.5.0. The four PT hardware filter is added to collect branch traces only from the regions of interest.

The offline tool is comprised of four parts: 1) the dynamic standard C library (*glibc* version 2.21) to intercept synchronization and memory allocation operations; 2) the modified *perf*

	Thread	Workload
apache	14	ApacheBench. 100K requests, 8 clients, 128KB file size
cherokee	38	ApacheBench. 100K request, 8 clients, 128KB file size
mysql	20	SysBench. 10K requests, 32 clients, 10 million records
memcached	5	YCSB. 200K requests, all ABCDE workload
transmission	4	1.48GB file
pfscan	4	6.8GB file
pbzip2	4	1GB file
aget	4	2.1GB file

Table 4.1: Evaluation Setup

tool to decode raw PT data; 3) the forward-and-backward replay engine that reconstructs memory traces, implemented using Intel's PIN [132] dynamic binary instrumentation tool; and 4) the FastTrack-based data race detector.

The PMU drivers and perf tool includes 4579 lines of C and assembly codes. The offline tools contain 7024 lines of C/C++ code, 793 lines of perl code, 105 lines of python code, and 623 lines of bash code.

4.6 Evaluation

This section evaluates ProRace's runtime overhead, trace size, data race detection effectiveness, memory reconstruction ratio, and offline analysis overhead.

4.6.1 Methodology

We ran experiments on a 4.0GHz quad-core Intel Core™i7-6700K (Skylake) processor, with 16GB of RAM, running Gentoo Linux Kernel 4.5.0. ProRace was evaluated using (1) PARSEC benchmark suite; and (2) seven real-world applications including *apache* web server,

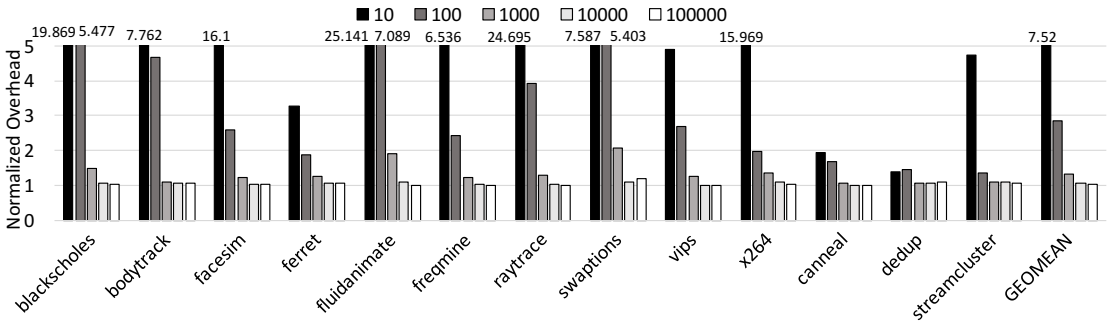


Figure 4.6: Performance overhead for PARSEC benchmarks

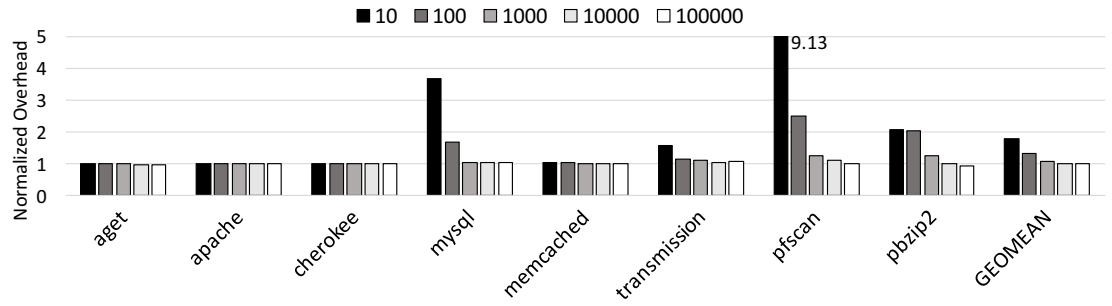


Figure 4.7: Performance overhead for real applications

mysql database server, *cherokee* web server, *pbzip2* parallel compressor, *pfscan* parallel file scanner, *transmission* BitTorrent client, and *aget* parallel web downloader. We use *simlarge* input for all the applications in the PARSEC suite and set the thread number to be four (equal to the number of cores). The evaluation setup for the real-world applications is listed in Table 4.1. All network and database applications were tested using the local area network which has a gigabit connection.

For data race detection analysis, ProRace was evaluated using 12 data race examples in real-world applications from previous study [226]. The 12 cases include three data races in *apache*, three races in *mysql*, two races in *cherokee*, two races in *pbzip2*, one race in *pfscan*, and the last one in *aget*. Some other cases in [226] are excluded because they do not include a data race, or are not well documented.

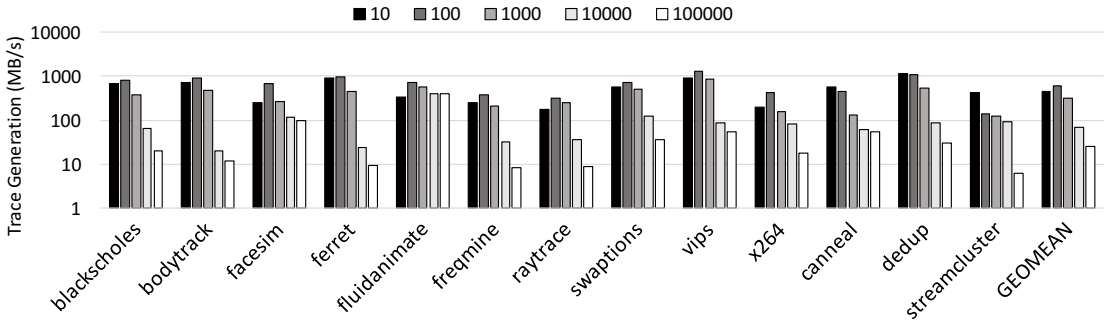


Figure 4.8: Space overhead for PARSEC benchmarks

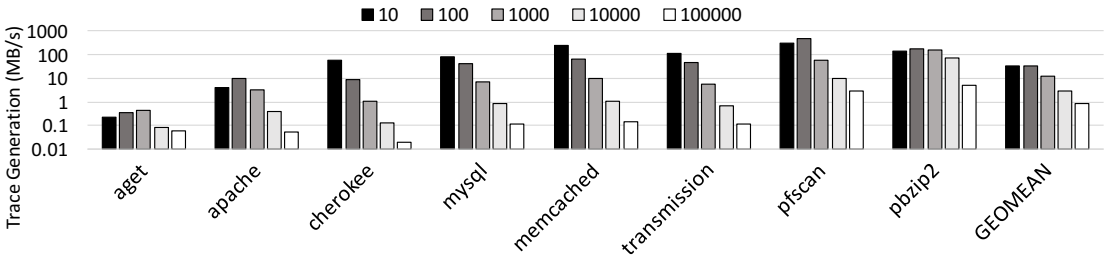


Figure 4.9: Space overhead for real applications

4.6.2 Performance Overhead

Figure 4.6 shows the performance overhead of ProRace for PARSEC benchmarks, with the varying PEBS sampling period from 10 to 100K. As expected, a small sampling period results in more samples, leading to high overhead. The geometric mean of performance overhead over all 13 applications in the PARSEC suite goes up from 4%, 7%, 31%, 2.85x, to 7.52x for the decreasing sampling period of 100K, 10K, 1K, 100, and 10, respectively. There are four applications *bodytrack*, *canneal*, *dedup*, *streamcluster* that incurs small 5-9% runtime overhead for the sampling period of 1K. Setting the sampling period to 10K makes 12/13 applications' overhead less than 10%. The user of ProRace can perform similar sensitivity analysis to find the lowest sampling period, given a performance overhead budget. Assuming the 10% budget, our experiment shows that the sampling period should be set between 1K and 10K for such CPU-intensive applications.

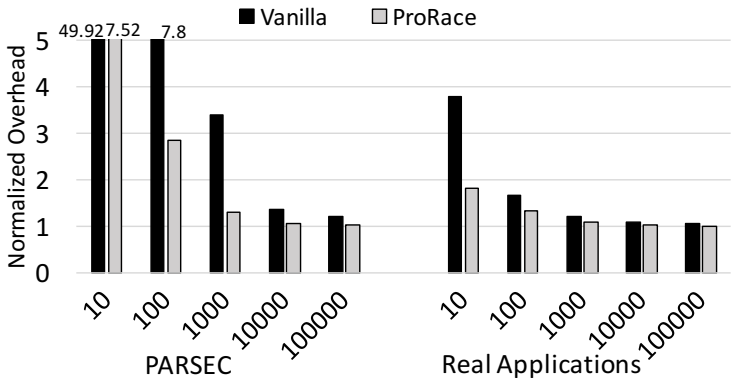


Figure 4.10: Performance overhead comparison

Figure 4.7 shows the performance overhead of ProRace for real world applications, with the varying PEBS sampling period from 10 to 100K. Some applications including *mysql*, *transmission*, *pfscan*, *pbzip2* showed a similar trend of high overhead for a small sampling period. However, the other applications shows negligible (<1%) overhead even with the very small sampling period of 10. The applications belonging to this second category are indeed network I/O dominant applications (with not much file I/O). The runtime overhead of ProRace can be hidden by network I/O. However, ProRace apparently cannot hide its overhead with file I/O well because it has to perform many writes to a file during tracing. On geometric average, the runtime overhead goes up from 0.8%, 2.6%, 8%, 34%, to 80% for the decreasing sampling period of 100K, 10K, 1K, 100, and 10, respectively. Assuming the 10% budget, our experiment shows that the sampling period may be set to smaller than 1K (even 10) for real (I/O-bound) applications.

The next study focuses on evaluating the efficiency of ProRace's new PEBS driver over the vanilla Linux driver. Figure 4.10 shows side-by-side runtime overhead comparing for each sampling period from 10 to 100K. For clarity, the figure only presents the geometric mean of PARSEC and real applications, respectively. As can be seen in the figure, ProRace's new driver outperforms the vanilla Linux driver. For an extreme case of the period of 10,

the vanilla driver incurs 50x slowdown, but ProRace shows 7.5x slowdown for PARSEC benchmarks. As another data point, with relatively large period of 100K, the vanilla driver incurs 20%, whereas the ProRace reports only 4% slowdown for the same benchmarks.

The overhead of RaceZ can be estimated to be around the same because it depends on the stock Linux driver. RaceZ also reports similar performance figures: 2.8% for the sampling period of 200K and 30% for 20K. The experimental result shows that ProRace has much less overhead than RaceZ. For example with the period 1K, RaceZ results in a 3.4x slowdown, whereas ProRace only incurs 31% overhead for the PARSEC suite.

As the last experiment for performance evaluation, we study a breakdown of runtime overhead among PEBS overhead, PT overhead, synchronization tracing overhead. We find that the PT overhead is very small contributing only 3% slowdown at most, respectively. The results show the benefits of PT's hardware supports for trace compression and memory range based filtering. Similarly, the synchronization tracing overhead also has a very small impact on performance (<1%). Finally, the PEBS overhead dominates the overall ProRace performance ranging from 97% to 99%. The result makes sense because PEBS events (memory operations) are much more frequent than PT records (branches), and PEBS events require rich information collection such as register states.

4.6.3 Trace Size

ProRace uses PEBS and PT to collect memory access samples and control-flow information at runtime. Figure 4.8 shows the trace size generated per a second during program execution of PARSEC benchmarks, with the varying PEBS sampling period from 10 to 100K. The PT trace size remains constant across different PEBS configurations, and its size is measured before decompression. As PT records are highly compressed by hardware, the PEBS trace

	Bug manifestation	Access Type	RaceZ			ProRace		
			Period:100	Period:1000	Period:10000	Period:100	Period:1000	Period:10000
apache-21287	double free	memory indirect	6	0	0	50	3	0
apache-25520	corrupted log	register indirect	14	3	0	57	52	15
apache-45605	assertion	register indirect	0	0	0	60	11	1
mysql-3596	crash	memory indirect	0	0	0	5	1	0
mysql-644	crash	memory indirect	20	1	0	21	6	1
mysql-791	missing output	memory indirect	12	0	0	59	2	0
cherokee-0.9.2	corrupted log	register indirect	43	11	2	63	29	8
cherokee-bug1	corrupted log	register indirect	7	3	0	57	19	5
pbzip2-0.9.4-crash	crash	memory indirect	0	0	0	0	0	0
pbzip2-0.9.4-benign	-	pc relative	2	0	0	100	100	100
pfscan	infinite loop	pc relative	0	0	0	100	100	100
aget-bug2	wrong record in log	pc relative	0	0	0	100	100	100
		(average)	8.7	1.5	0.2	56	35.3	27.5

Table 4.2: Data Race Detection

dominates the overall trace size (∼99%). As expected, a small sampling period results in more samples, leading to large trace size. Note that the y-axis is logarithmic. On geometric average, the trace size per second (in MB/s) goes up from 26, 69, 321, 597, to 463 for the decreasing sampling period of 100K, 10K, 1K, 100, and 10, respectively. One outlier is that the trace size for the sampling period of 10 turns out to be less than that of 100 (though it incurs higher overhead as shown in the above experiment). Further investigations show that with a very low sampling period, though the hardware may sample more, these samples may be dropped if the kernel finds that too much time has been spent on the interrupt handling. This implies that there is no benefit of setting the sampling period smaller than a certain (application-specific) threshold.

Figure 4.9 shows the trace size per second (in MB/s) for real-world applications, with the varying PEBS sampling period from 10 to 100K. The result shows the similar trend but much less space overhead compared to PARSEC benchmarks. On geometric average, the trace size per second (in MB/s) goes up from 0.2, 1.2, 7.9, 40.8, to 99.5 for the decreasing sampling period of 100K, 10K, 1K, 100, and 10, respectively.

4.6.4 Race Detection

To evaluate the ProRace's effectiveness in data race detection, we used 12 real-world data race bugs [226]. For each race bug, we fed a buggy input as documented in the previous study [226], and did not control the thread schedules. We collected 100 traces for each PEBS sampling period: 100, 1K and 10K; and counted how many times ProRace can report the data race among the 100 traces. In effect, the resulting number can be regarded as an approximate detection probability. For comparison, we also measured the number of data races detected by RaceZ. Note that RaceZ enables memory trace reconstruction within one basic block, and for backward replay, it only supports a trivial form of backward propagation within that single basic block. On the other hand, ProRace includes PT-based full forward-and-backward replay across basic blocks; and supports backward propagation and reverse execution based backward replay.

Table 4.2 shows the summary of ProRace's data race detection effectiveness. The first column corresponds to the application name and its bug-tracking number, if exists, while the second refers to how the bug manifests during a program execution. The third column describes its characteristics that we analyzed manually. The next six columns show the number of traces where RaceZ and ProRace detect data races out of 100 traces (i.e., representing the detection probability) for each sampling period of 100, 1K and 10K, respectively.

It is important to note that ProRace does detect a data race. As expected, in general, the detection probability increases as the sampling period decreases. On the other hand, some race bugs in *pbzip2-0.9.4*, *pfscan*, and *aget-bug2* are detected every time (100%). Examining the results, we see that the address of the racy variable uses PC-relative addressing in the program. Thus, reproducing the address of such racy memory accesses is easy because the `%rip` register is always available as PC, i.e., an instruction pointer. Here, to detect such

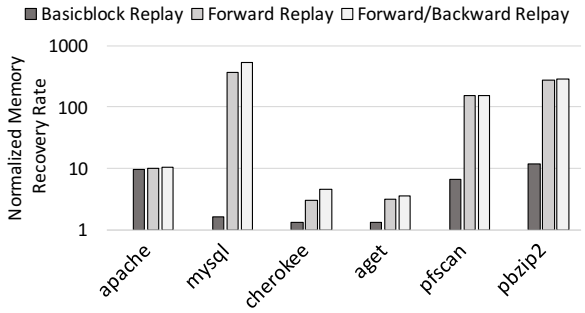


Figure 4.11: Memory Recovery Ratio

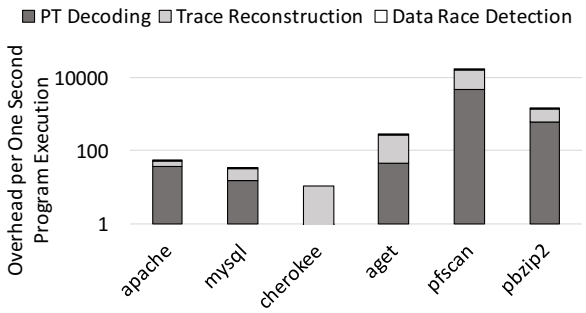


Figure 4.12: Offline analysis overhead

race bugs, ProRace only needs to know what basic blocks contain the racy memory accesses, which is obtained by PT's control-flow trace, without understanding the PEBS-provided execution contexts.

As can be seen, for a given sampling period, ProRace detects many more data races than RaceZ. For example, ProRace improves the detection probability from 0.2% to 27.5% on average (arithmetic mean) for the sampling period of 10K, which only incurs 2.6% runtime overhead (Figure 4.7). For the low sampling period of 100, ProRace can detect almost all cases (11/12), but RaceZ misses many. It also turns out that RaceZ cannot effectively detect races on simple PC-relative addressing cases because RaceZ requires sampling at the exact basic block containing the racy access. Overall, the results show that ProRace's PT-guided forward-and-backward replays are very helpful in detecting data races.

4.6.5 Memory Operation Reconstruction

ProRace leverages the forward and backward replays to reconstruct unsampled memory operations. RaceZ also tries to recover other memory accesses, but its scope is limited to one basic block that the sampled instruction belongs to. This section shows the benefit of using ProRace's forward and backward replays in terms of the memory reconstruction ratio.

Figure 4.11 shows the memory instruction recovery ratio (i.e., the number of recovered and sampled memory operations normalized to the number of original PEBS-sampled instructions) for the six buggy applications with the sampling period of 10K.

The first left-most bar shows how many more memory operations can get reconstructed within one basic block (equivalent to RaceZ's approach). The results show that the basic-block granularity recovery scheme can reconstructs only 1.3x-11.9x memory operations, with the average (arithmetic mean) of 5.4x ratio. Upon further investigation, we found out that *apache*, which shows a good 9.53x recovery ratio, has a lot of simple memory instructions that use PC-relative addressing in a basic block. However, that was not the case for other applications like *mysql*, which shows only a 1.6x recovery ratio.

The next two bars show the benefit of forward replay only and forward+backward replays in ProRace. On average, the forward replay recovers 134x more memory accesses compared to the baseline (PEBS samples). The backward replay provides additional benefits, and when the backward replay is combined with the forward replay, they achieve a higher recovery ratio of 164x on average. The results shows that ProRace's race detection coverage (which is approximately proportional to the number of recovered and sampled memory operations) is more than 30 times better than RaceZ's limited basic-block level reconstruction.

4.6.6 Offline Analysis Overhead

Lastly, Figure 4.12 shows the offline analysis overhead when traced with the sampling period of 10K. The results shows that to analyze one second of program execution, offline analysis takes 54.5 seconds for *apache* and 35.3 seconds for *mysql*. *Pfscan* shows the worst analysis overhead as it generates a very large trace for a short amount of program execution time.

We also present the breakdown of offline analysis overhead, including 1) PT trace decoding;

2) memory trace reconstruction; and 3) data race detection, each of which takes 33.7%, 64.7%, and 1.6% of the total offline analysis cost, respectively (note the logarithmic y-axis). Note that we conducted this experiment using a single machine. However, the PT trace decoding and trace reconstruction parts, which contributes >98% of the total cost, can be easily parallelized. PT records are independent each other, and the forward-and-backward replay can be also performed region by region, making it suitable for using multiple analysis machines. Moreover, the result includes the overhead of PIN-based dynamic binary instrumentation. The same features can be implemented using static instrumentation tools [116, 229] for better performance.

4.7 Summary

In this chapter we repurpose PMU hardware, which is originally designed for performance profiling (**TS1**), for lightweight data race detection (**PS1**). We presents ProRace, a novel PMU sampling-based data race detector that can be deployed in production settings. Its new kernel driver, that eliminates unnecessary copying and data processing, significantly lowers the run-time overhead of using PEBS to sample memory accesses. Furthermore, ProRace introduces a novel technique to reconstruct unsampled memory operations with the PT-guided forward and backward replays, thereby enhancing the data race detection coverage. The experimental results highlight ProRace's high data race detection capability using the 12 real-world data race bugs. ProRace is published at [231].

Chapter 5

Memory Efficient Temporal Memory Safety Solution for MPX

In previous chapters (Chapter 3 and Chapter 4), we discussed leveraging commodity hardware to reduce run-time overhead of dynamic data race detectors. In this chapter, we focus on memory safety bugs which are more common than data races and try to solve space overhead problem of memory safety bug detectors (**PS2**). Memory safety bug often cause security vulnerability, recent Intel processors support hardware-accelerated bound checks, called Memory Protection Extensions (MPX). Unfortunately, MPX provides no temporal safety. In this chapter, we presents *BOGO*, a lightweight full memory safety enforcement scheme that transparently guarantees temporal safety on top of MPX's spatial safety. Instead of tracking separate metadata for temporal safety, *BOGO* reuses the bounds metadata maintained by MPX for both spatial and temporal safety. On `free`, *BOGO* scans the MPX bound tables to invalidate the bound of dangling pointers; any following use-after-free error can be detected by MPX as an out-of-bound error. Since scanning the entire MPX bound tables could be expensive, *BOGO* tracks a small set of hot MPX bound table pages to check on `free`, and relies on the page fault mechanism to detect any potentially missing dangling pointer, ensuring sound temporal safety protection.

Our evaluation shows that *BOGO* provides full memory safety at 60% runtime overhead and 36% memory overhead for SPEC CPU 2006 benchmarks. We also show that *BOGO*

incurs a reasonable 2.7x slowdown for the worst-case malloc-free intensive benchmarks; and moderate 1.34x overhead for real-world applications.

5.1 Introduction

Memory unsafe languages such as C/C++ are prone to bugs leading to memory safety violations [192]. Spatial safety violation occurs when memory access is not within the object's bound (e.g., buffer overflow), while temporal safety violation[1] happens when accessing a deallocated object (e.g., use-after-free).

Many security breaches are caused by memory safety violations [5]. While buffer overflow vulnerabilities have been exploited for return oriented programming [161] and other code reuse attacks [44, 53, 197], use-after-free vulnerabilities have also been exploited to corrupt control flow: e.g., virtual function table hijacking [175].

Keeping pace with a broad range of research, hardware support for security has been adopted in mainstream commodity processors – notably, Intel's Memory Protection Extensions (MPX) for hardware-accelerated bounds checking. MPX keeps track of per-pointer bound metadata (the base and bound of heap/stack objects) in bound tables. On a pointer dereference, MPX checks if the pointer value remains within the bounds, ensuring spatial memory safety. Unfortunately, MPX does not support temporal memory safety. Thus, full memory safety can only be achieved by augmenting MPX with a separate temporal safety solution. However, existing solutions for temporal memory safety require their metadata tracking and checking. DangSan[199], DangNull[117], Undangle[49] and FreeSentry[225] maintain designated data structures for temporal safety and do pointer nullification at the time of

[1]We use "temporal memory safety" and "no use-after-free vulnerabilities" interchangeably, though the former subsumes the latter. The same is true for "spatial memory safety" and "no out-of-bound vulnerabilities".

free(). The dereference of a dangling pointer is discovered as a segment fault. CETS[145] also maintains a separated metadata for each pointer and memory object. The detection is achieved through explicit check. Simply combining existing temporal memory solutions with Intel MPX would double the time and space overhead.

We propose a lightweight full memory safety enforcement scheme, BOGO (Chapter 5), that works with MPX (Intel Skylake onwards). This work demonstrates a novel software solution that transparently extends MPX to support both spatial and temporal memory safety, without additional hardware support or significant performance degradation.

The key insight to realize the "promotion" is that the MPX bound table can be searched for dangling pointers to an object when it is freed. Since the bound table entry already maintains the bound information for each pointer, dangling pointers can be identified by checking for bounds enclosing the address being freed. For each dangling pointer `p` found, it invalidates the bound information of the bound table entry, indexed by that pointer `p`. On the later dereference of `p` (use-after-free), MPX instrumentation will find an invalid bound (as a part of its bound checking), and raise an exception. In effect, it achieves temporal safety by transforming it into spatial safety.

This approach relieves it of the burden to maintain and check a separate temporal memory safety metadata, reducing time overhead and more drastically space overhead. it introduces a new synergistic way to enforce spatial and temporal memory safety by repurposing one for another. However, scanning the entire MPX bound tables on each `free` could lead to significant performance overhead. it leverages a novel page-protection-based technique to address this performance challenge. it tracks the working set of MPX bound table pages and only searches those *hot* pages on `free` for performance. To track a dangling pointer potentially in the rest *cold* pages, it makes the cold MPX bound table pages non-accessible. Any following access to a cold page is always preceded by a page fault. its page fault handler

scans the faulted MPX page and invalidates any dangling pointers therein, guaranteeing soundness.

This work makes the following contributions:

• To the best of our knowledge, *BOGO* is the first temporal memory safety protection solution that does not maintain its own metadata, but seamlessly reuses bound metadata tracked for spatial memory safety.

• *BOGO* transparently provides an MPX-enabled binary with full memory safety without application change or other hardware support.

• We implement `llvm-mpx`, an LLVM-based MPX pass, with sound bound checking optimizations, outperforming existing MPX compilers.

• The experimental results show that *BOGO* can support full memory safety at comparable (in many cases, better) runtime overhead and much less memory overhead, compared to the state-of-the-art solutions.

• We stress-test *BOGO* with the worst-case malloc-free intensive benchmarks, and also evaluate *BOGO*'s interoperability and scalability for real-world multithreaded applications.

5.2 Overview of *BOGO*

The goal of this project(*BOGO*) is to provide full memory safety on top of MPX-enabled processors without significant overheads. With *BOGO*, users can buy such processors for spatial memory safety, and get temporal memory safety (almost) free; hence relieving the burden of the compiler and architectural support for temporal memory safety guarantee. More precisely, this paper focuses on temporal memory safety for heap objects (use-after-

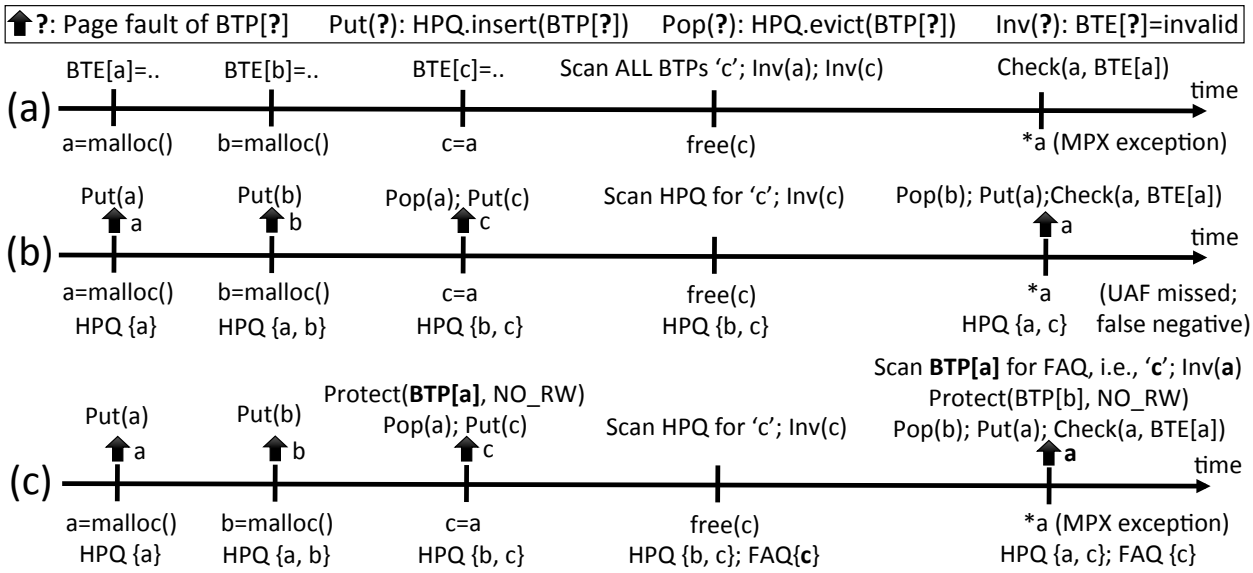


Figure 5.1: BOGO: FullScan/PartialScan and PageFaultScan
(a) FullScan; (b) PartialScan only; and (c) PartialScan and PageFaultScan. (b) misses the use-after-free error, and (c) solves the problem with PageFaultScan. For brevity, (b) and (c) omit the bound creation and propagation. A bold uparrow represents a BTP fault. `Put` and `Pop` inserts/evicts a BTP to/from HPQ. `Inv` invalidates a BTE.

free), and *BOGO* in the current form does not provide temporal safety for stack objects (use-after-return). Additional support required for stack objects is discussed in Section 5.6.

Threat Model and Assumptions. *BOGO* relies on MPX for spatial memory safety, and adds temporal memory safety upon it. Therefore, *BOGO* assumes that underlying MPX-enabled processors can be trusted, and there are no hardware security bugs in the processor circuit fabrication [209, 220]. This work also assumes that adversaries cannot corrupt the MPX metadata by using non-memory-safety related attacks such as row hammer attacks [178, 200] or illegitimately having higher (root) privilege. Any attempts to corrupt the MPX metadata by exploiting memory safety vulnerabilities will be detected by *BOGO* itself. As *BOGO* relies on the soundness of MPX metadata, we further assume that MPX instrumentation is applied to all the source codes when soundness is required.

Overview. *BOGO* takes binary compiled with MPX instrumentation and transparently

achieves temporal memory safety by reusing MPX metadata. At a high level: On `free` of a pointer `p`, it searches BTs for the entry whose bound overlaps with the object being deallocated. The existence of such a BTE implies that another pointer, say `q`, also pointing to the same object, becomes a dangling pointer. When found, it invalidates the metadata of dangling pointers. Later dereference of pointer `q` will be checked by MPX (for the default spatial memory safety), leading to an `OutOfBound` exception because of the invalidated bound. The beauty of *BOGO* is that it enforces temporal memory safety by triggering the violation of spatial memory safety. Users can differentiate temporal from spatial violations by checking a special value in the bound register.

BOGO attempts to eliminate dangling pointers on `free` like the aforementioned pointer graph-based temporal memory safety solutions. However, there is one big difference. *BOGO* does not maintain additional metadata (e.g., pointer graph) for temporal memory safety. Instead, it reuses MPX metadata as is, and scans the BTs to invalidate dangling pointers.

FullScan. Consider an example in Fig. 5.1(a). On `free(c)`, a naive FullScan checks all BTs, finds aliased pointer `a`, and invalidates `BTE[a]` so that later use of dangling pointer `a` would result in an MPX exception. However, searching the whole MPX BTs can lead to unacceptable performance degradation due to the large search space, thus care must be taken to minimize the cost.

PartialScan. To bound the scan cost on `free`, *BOGO* tracks a small set of hot, recently used BTPs (Bound Table Pages) using a page protection mechanism, keeps them in the *Hot BTP Queue* (HPQ), and performs PartialScan that looks for dangling pointers only on the hot BTPs in the current HPQ.

Fig. 5.1(b) shows an example. Suppose the bounds of pointers `a`, `b`, and `c` are stored in different BTPs: `BTP[a]`, `BTP[b]`, and `BTP[c]`. Further assume that the size of HPQ is 2.

Each page fault on the first pointer access causes the corresponding BTP to be inserted into HPQ. When HPQ becomes full on the `BTP[c]` page fault (at the statement `c=a`), *BOGO* evicts the cold (least recently added) `BTP[a]` from the HPQ and inserts the hot `BTP[c]` into the HPQ. Then, `free(c)` checks dangling pointers only against `BTP[b]` and `BTP[c]` in the current HPQ, a small subset of all BTPs.

However, this PartialScan may lead to a false negative (i.e., missing a use-after-free error) when a dangling pointer happens to be in a cold BTP. In Fig. 5.1(b), as `BTP[a]` is not in HPQ, `BTE[a]` was not scanned/invalidated by `free(c)`, even though `a` and `c` are aliased. Thus, the following dereference of the dangling pointer, `*a`, remains undetected.

PageFaultScan. *BOGO* introduces PageFaultScan to guarantee sound temporal memory safety. When a BTP is evicted from HPQ, *BOGO* marks the cold BTP not-readable and not-writable so that the later access to the cold BTP can be trapped by a page fault. Note that when a pointer is dereferenced, MPX always accesses its BTE for the spatial memory safety check, resulting in a page fault. Meanwhile, on `free`, *BOGO* also tracks the freed addresses in the Freed Address Queue (FAQ) if they are partially checked only over hot BTPs: i.e., by PartialScan, not by FullScan. Upon a page fault of a cold BTP, *BOGO* checks against freed addresses in FAQ to see whether there exists any dangling pointer to the addresses in the cold BTP. Then, *BOGO* recovers page access permissions of the BTP and puts it into the HPQ.

Using the same example, Fig. 5.1(c) illustrates that *BOGO* guarantees the detection of all use-after-free errors using both PartialScan and PageFaultScan. Though PartialScan on `free(c)` did not invalidate the bound of `BTE[a]`, the dereference of pointer `a` would result in a page fault on which PageFaultScan can detect the use-after-free error by scanning the `BTP[a]` for the freed address `c` stored in FAQ.

5.3 *BOGO* Approach Details

This section presents how *BOGO* tracks hot BTPs to bound the scan cost on `free` (Section 5.3.1); how PartialScan and PageFaultScan can ensure no false negative (Section 5.3.2); and how to achieve redundancy-free and false-positive-free PageFaultScan (Section 5.3.3).

5.3.1 Hot Bound Table Page Tracking

Fig. 5.2 (Lines 2-12) shows how *BOGO* makes use of a page protection mechanism to track hot BTPs at a low cost. Upon a BTP fault, *BOGO* restores the read/write permissions (Line 3) and puts this "hot" (most recently accessed) BTP into the bounded Hot BTP Queue (HPQ) (Line 9). When the HPQ is full, *BOGO* evicts the "coldest" (least recently added) BTP in a FIFO manner, and makes it not-readable and not-writable (Lines 9-12). The latter part of this section discusses the rest of the BTP fault handler.

5.3.2 Combine PartialScan and PageFaultScan to Achieve Low Overhead and No False Negative

On `free`, *BOGO* scans BTs and invalidates the BTE of dangling pointers. Fig. 5.2 (Lines 14-21) presents how *BOGO* instruments `free`. *BOGO* can safely rely on PartialScan and PageFaultScan as long as it can hold free addresses in the FAQ. In some cases, the FAQ can be configured to be large enough to avoid FullScan. In other cases, if the FAQ becomes full, *BOGO* falls back to FullScan that checks all the free addresses in FAQ over all the BTs (Line 20). We discuss FullScan optimization in Section 5.4.2. After FullScan, *BOGO* may reset FAQ (Line 21) as there are no longer pending temporal memory safety checks to perform. Any unused BTs can be safely reclaimed at this point. This approach trades performance

for soundness.

For performance, *BOGO* favors PartialScan (Line 18) that looks up dangling pointers only over hot BTPs in the HPQ. To ensure soundness (see the difference between Fig. 5.1(b) and Fig. 5.1(c)), the free addresses that are checked via PartialScan are collected in the Free Address Queue (FAQ) along with the free time[2] (Line 16). These free addresses remain in the FAQ until the next FullScan (Line 20-21), and meanwhile they are checked over the (cold) BTPs resulting in page faults via PageFaultScan (Line 4-8). The `evict_time` and `free_time` will be discussed in the next section.

5.3.3 Combine PageFaultScan and RedundancyPredication to Achieve Low Overhead and No False Positive

Though PageFaultScan ensures no false negatives, it may lead to a false positive. Consider the following code:

```
a=malloc(8); b=malloc(64); c=a; free(c);
a=malloc(8); *b; c=b; free(b); *a;
```

The pointer `a` was once a dangling pointer after `free(c)`, but is reassigned by the second `a=malloc(8)` of the same size, rendering `*a` legal. However, if this `malloc` reuses a freed object for locality, which is the case for modern allocators, PageFaultScan may raise a false alarm.

Figure 5.3(a) illustrates the case with the heap snapshot change over time. At time `t1`, `a=malloc(8)` returns `0x10` and sets `BTE[a]`. At `t3`, `BTP[a]` is evicted from HPQ. At `t4` on `free(c)`, PartialScan does not check `BTP[a]`, and puts the freed address (`0x10`) in FAQ. At

[2]This is an abstract time that we implement as the FAQ index, to avoid the cost of using real timestamps.

```
1  /* [btp] and [faddr] form a single element list with the parameter */
2  OnBoundTablePageFault(btp)
3    mprotect(btp,RW)
4    evict_time = get_evict_time(btp)
5    for each faddr in FAQ
6      free_time = get_free_time(faddr)
7      if (evict_time < free_time)
8        scan([btp],[faddr]) // PageFaultScan
9    evicted_btp = insert(HPQ,btp)
10   if (evicted_btp != NULL)
11     set_evict_time(evicted_btp)
12     mprotect(evicted_btp,NONE)
13
14 OnFree(faddr)
15   free(faddr)          // actual free
16   insert(FAQ,faddr)    // index as free time
17   if (FAQ.length != MAX)
18     scan(HPQ,[faddr]) // PartialScan
19   else
20     scan(ALL,FAQ)      // FullScan
21     reset(FAQ)
22
23 scan(btp_list,faddr_list)
24   for each btp in btp_list
25     for each faddr in faddr_list
26       // scan and invalidate if overlaps
27       for each bte in btp
28         if(bte.base<=faddr && faddr<=bte.bound)
29            bte.base = INVALID
30            bte.bound = INVALID
```

Figure 5.2: *BOGO* handler algorithms.

`t5`, the second `a=malloc(8)` happens to return the same location `0x10`, as shown in the third heap snapshot. At `t7`, `BTP[a]` becomes cold again. As a result, the last `*a` at `t9` results in a page fault on `BTP[a]`. PageFaultScan finds an overlap between `BTE[a]` and `0x10` in FAQ. However, this is a false alarm.

One naive workaround would be not to release the memory to the system on `free` so that the freed object cannot be reused for later allocation, until *BOGO* performs FullScan. However,

obviously, it will increase the memory footprint significantly.

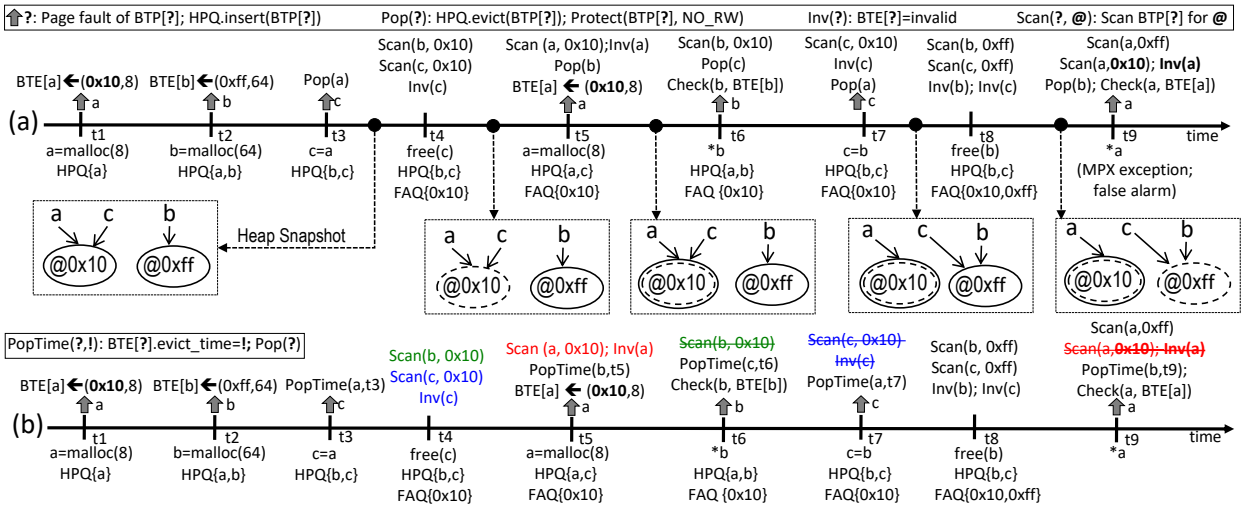


Figure 5.3: Redundancy prediction: (a) shows a false positive case, and (b) shows how *BOGO* removes the redundant scans and eliminates the false positive. Actions of *BOGO* appears above the bar while the status of HPQ/FAQ does underneath the code. Each color represents a different redundant scan.

Redundancy Predication. We present a novel *redundancy prediction* technique. The crux of the problem is due to redundant checks. Focus on the three cases: `t4` when PartialScan adds the freed address `0x10` into the FAQ; `t7` when `BTP[a]` becomes cold; and `t9` when PageFaultScan performs a check on the freed address `0x10`. Recall that we originally introduced PageFaultScan because PartialScan did not perform a check on cold BTPs at that time. However, in this scenario, `BTP[a]` becomes cold after a PartialScan, implying that PageFaultScan does not need to check the freed address `0x10` on `BTP[a]`. More formally speaking, it is safe for PageFaultScan to skip the scanning of BTP for a given freed address `faddr` in FAQ if $Time_{evict}^{BTP}$ is greater (i.e., later) than $Time_{freed}^{faddr}$. We provide the proof at the end of this section.

Based on this observation, *BOGO* keeps track of $Time_{freed}^{faddr}$ when `faddr` is added in FAQ (Line 16), and $Time_{evict}^{BTP}$ when a `BTP` is evicted from HPQ (Line 11). On PageFaultScan, *BOGO* compares their times (Lines 4-7), and avoids redundant checks, leading to better

performance and no false positives.

To illustrate, Figure 5.3(b) shows how *BOGO* deals with the false positive with redundancy prediction. Since the eviction time of the `BTP[a]` (`t7`) is greater than the freed time of `0x10` (`t4`), PageFaultScan (`t9`) safely skips `Scan(a,0x10)` and avoids the false alarm. For the same reason, *BOGO* skips PageFaultScan at times `t6, t7` for `BTP[b]` and `BTP[c]`, respectively. However, PageFaultScan at `t5` cannot be skipped, and it needs to check the faulted page `BTP[a]` with the freed address `0x10`. Note, for all the scans that can be safely skipped, they are paired with the corresponding previous scan that performs the same bound check. Such a pair is shown using the same color in Fig. 5.3(b), where three pairs exist (green, blue, and red).

Now we prove that this redundancy elimination is safe.

Theorem 1. On a BTP fault, it is safe (no missing detection) for PageFaultScan to skip the scanning of the BTP for a given freed address *faddr* in FAQ if $Time^{BTP}_{evict}$ is larger than $Time^{faddr}_{freed}$.

Proof. We provide a direct proof sketch. Recall *BOGO* basically has two scan methods, PageFaultScan and the free time PartialScan which checks HPQ. Thus, we need to prove that either method has already scanned the BTP that satisfies the time condition, i.e., $Time^{faddr}_{freed} < Time^{BTP}_{evict}$. First, if the BTP was a part of HPQ at $Time^{faddr}_{freed}$, the `free` must have scanned the BTP obviously. Thus, this case makes it redundant to scan the BTP in the current PageFaultScan (referred to as *currPFS*).

Second, if PartialScan on `free` did not scan the BTP (i.e., it was not in the HPQ at $Time^{faddr}_{freed}$), it must have been evicted before; let's refer to the time as $Time^{BTP}_{pastEvict}$. The implication is there must be a PageFaultScan (referred to as *pastPFS*) between two evictions, and it must have happened after the `free` which otherwise would have scanned the BTP, and we get the

following:

$$\begin{aligned} Time^{BTP}_{pastEvict} &< Time^{faddr}_{freed} < Time^{BTP}_{pastPFS} \\ &< Time^{BTP}_{evict} < Time^{BTP}_{currPFS} \end{aligned} \tag{5.1}$$

Then, we investigate if the *pastPFS* scanned the BTP. As shown above inequality, for the *pastPFS*, the eviction time of the BTP ($Time^{BTP}_{pastEvict}$) is not larger than the free time ($Time^{faddr}_{freed}$). Thus, *pastPFS* must have scanned the BTP, making it redundant to scan the BTP in *currPFS*. Consequently, Theorem 1 must be true. □

In sum, with PageFaultScan and redundancy prediction, *BOGO* can eliminate unnecessary scans, achieving better performance and no false positives.

5.4 Optimization

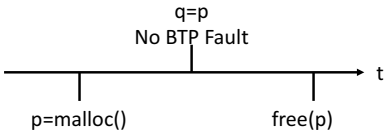


Figure 5.4: Example of sound PartialScan. No further PageFaultScan is required.

	Partial Scan		Page Fault Scan		Full Scan	
	number of scans	per-scan cost $O(\|HPQ\|)$	number of scans	per-scan cost $O(\|FAQ\|)$	number of scans	per-scan cost $O(\|ALL\| * \|FAQ\|)$
HPQ ↑	-	↑	↓	-	-	-
FAQ ↑	↑	-	-	↑	↓	↑

Table 5.1: Impacts of increasing HPQ and FAQ on the number and the cost of Partial, Page Fault, and Full Scans.

5.4.1 No PageFaultScan Optimization

On `free`, when FAQ is not full, *BOGO* performs PartialScan that checks hot BTPs and stores the freed address into FAQ so that later PageFaultScans can detect dangling pointers that PartialScan might miss (Section 5.3.2). This PageFaultScan backup mechanism is necessary because dangling pointers may have resided in cold BTPs. If *BOGO* can prove the absence of dangling points in the cold BTPs, then it does not need to add the freed address into the

FAQ, bringing two benefits: 1) to save the FAQ space (triggering FullScan slowly) and 2) to avoid succeeding PageFaultScans against the freed address.

Consider an example in Fig. 5.4 where there were no BTP faults between the memory allocation and deallocation. The absence of BTP faults implies that any potential copy of pointer `p`, the necessary condition of dangling pointers, must have happened only with those pointers whose BTEs lie in the hot BTPs. Otherwise, a BTP fault would have been triggered and HPQ has been altered. In this case, on `free`, *BOGO* only needs to check the freed address with the hot BTPs in the current HPQ, and there is no need to add it to the FAQ for later PageFaultScans. In practice, applications often have short-living heap objects where `malloc` and `free` are adjacent to each other in time.

To support this optimization, *BOGO* maintains a small hash table which tracks the addresses of objects that have been allocated since the last BTP fault. *BOGO* stores the address being allocated on `malloc`, and checks if the hash table holds the address being deallocated on `free`. When there is a match, PartialScan applies the proposed optimization by not putting the freed address into the FAQ. *BOGO* resets the hash table on a BTP fault (as a part of HPQ maintenance).

5.4.2 FullScan Optimization

When FAQ is full, *BOGO* performs FullScan over the entire BTs (Fig. 5.2 Line 20). A naive implementation would iterate over the (1^{st}-level) BD to find all the valid (2^{nd}-level) BTs. Scanning the huge BD leads to severe performance degradation. Even for a valid BT, many of its BTPs may have not been accessed, and thus scanning them would cause unnecessary page faults. To avoid scanning the BD and all BTs, which would be an order of magnitude slower, *BOGO* uses a custom syscall to get only the accessed BTPs, and scans them directly.

The syscall looks up per-process memory descriptor for virtual memory area(VMA) reserved for MPX and returns those pages whose *accessed bit* is set in the page table.

5.5 Dynamic Adaptation of Queue Size

BOGO maintains two queues: HPQ and FAQ. Their sizes have a significant impact on the number and cost of Partial, Page Fault, and Full Scans that determine the overall performance. This section first analyzes the cost of each scan (Section 5.5.1) and the impacts of HPQ and FAQ sizes (Section 5.5.2). Then, we introduce *scan cost*-based dynamic adaptive scheme that adjusts the size of HPQ at runtime for optimal performance (Section 5.5.3).

5.5.1 Scan Cost Analysis

In general, the cost of each scan is the product of the number of BTPs to scan and the number of free addresses to scan (Fig. 5.2 Lines 24-30). Note that the innermost loop in Line 27 iterates over (constant number of) 128 BTEs (of size 32B each) in a BTP (of size 4KB). Therefore, the cost of PartialScan is $O(|HPQ|)$ as it scans the Hot BTPs in the HPQ against a single pointer address being freed. The cost of PageFaultScan is $O(|FAQ|)$ as it checks the freed addresses in the FAQ against a single (once cold now hot) BTP being page faulted. Lastly, the cost of FullScan is $O(|ALL| * |FAQ|)$ as it checks the freed addresses in the FAQ against all BTPs.

5.5.2 Impact of HPQ and FAQ Sizes

Table 5.1 summarizes the impacts of increasing the size of HPQ and FAQ on the number and the cost of each scan, while decreasing its sizes has an opposite effect.

Increasing HPQ. HPQ keeps track of Hot BTPs to perform PartialScan, and thus increasing the size of HPQ would increase the cost of PartialScan. The number of PartialScans is irrelevant to the size of HPQ. To be precise, the number depends on the `free` frequency and the size of FAQ (as it determines which Partial or Full Scan to take on `free`). On the other hand, increasing HPQ would decrease the number of PageFaultScans as HPQ can hold more hot BTPs. Note that the total cost of each scan would be proportional to the number of scans and the per-scan cost. The size of HPQ has opposite impacts on these two scans. Therefore, for the common cases where PartialScan and PageFaultScan are used (without FullScan), the size of HPQ should be tuned to make a good balance on both scans. Our sensitivity study on the HPQ size in Section 5.8.3 shows that each application has a different optimal HPQ size, motivating our adaptive scheme in Section 5.5.3.

Increasing FAQ. Unlike HPQ, the bigger FAQ, in general, leads to better performance. First, its impacts on PartialScan is small because the number of PartialScans varies slightly. Second, increasing FAQ at a glance may look like harming PageFaultScan as it iterates over the free addresses in the FAQ. However, in reality, this is not true because PageFaultScan stops scanning when it finds a freed address whose free time is earlier than the eviction time of the BTP being page faulted as discussed in Section 5.3.3 and Fig. 5.3. In other words, the cost of PageFaultScan even with the very large FAQ is in effect bounded. Lastly, as the cost of FullScan is way more expensive than the other two scans, it is better to keep its number low by making the FAQ big enough. Therefore, in the next section, we focus on tuning the HPQ size at runtime.

5.5.3 Scan Cost-based HPQ Adaptive Scheme

Based on the above observation, *BOGO* dynamically adjusts the size of HPQ to balance PartialScans and PageFaultScans. To this end, *BOGO* divides a program execution into regular intervals, called *quanta*. Then, at runtime *BOGO* measures and compares the cost (execution time) of PartialScans and PageFaultScans in a quantum, and then decides whether to reduce or enlarge the size of HPQ for the next quantum. If the cost of PageFaultScan is higher than that of PartialScan, *BOGO* increases the HPQ, and vice versa. This scan cost based adaptive scheme allows *BOGO* to adapt application specific characteristics (e.g., `free` frequencies) and the program phase changes even within the same application. Our experimental results in Section 5.8.3 show that *BOGO* with adaptive HPQ outperforms the one with profiled-based manual configuration. By default, *BOGO* uses the quantum of 100 ms, sets the initial size of HPQ to be 16, and changes the HPQ size exponentially.

5.6 Discussion

Free-after-free. Consider the following code:

```
a=malloc(8); b=a; free(a); a=malloc(8); free(b);
```

Suppose two mallocs are allocated to the same region. Then, `b` would free `a`'s buffer. If one sees `free(b)` as the use of `b`, *BOGO* can perform a bound check on `free(b)`. As *BOGO* invalidates the `BTE[b]` on `free(a)`, it can detect such a case.

Use-after-return. This refers to dereference of deallocated stack object [84]. *BOGO* can be extended to detect it by invalidating the bounds belonging to current stackframe upon return. Static analysis can help to avoid such a check in many cases, thus supporting the detection at a low cost. For instance, static analysis can tell whether there is a pointer that

points to the stackframe and escapes the function. Such cases are expected to be rare: (1) a pointer to the current stackframe is returned; (2) a pointer to the stackframe is propagated across the function. It implies that for most of returns, the scan could be avoided.

Multithreading. Metadata-based memory error detectors may lead to false positives or false negatives when a pointer operation and metadata updates/checks do not happen in an atomic manner. For solutions like CETS [145] that only lookup, update, and check per-pointer metadata at a time, if a program is data-race-free, atomicity could be achieved by placing instrumentation codes into the same critical section as the original pointer operation. For a program with data races, this remains a challenge. On the other hand, for solutions like DangSan [199] and *BOGO*, that access other pointer's metadata (for invalidation), the problem gets worse because concurrent metadata updates from independent pointers may form a race condition: e.g., while one scans on `free`, another may update the metadata. DangSan chooses to favor performance over soundness without additional support. The current *BOGO* prototype shares the same limitation. However, it is possible to mark the BTPs to be scanned as non-accessible during scanning and make a concurrent thread wait at the page granularity (instead of stopping the world). This design remains future work. During our experiments with multithreaded applications (Section 5.8.5), we did not observe false warnings (false negatives are unknown).

Custom Memory Allocator. They need to be patched to invoke *BOGO*'s scan, which otherwise may lead to false negatives. They can be identified by using techniques like [55].

5.7 Implementation

The llvm-mpx pass consists of 9240 LoC, along with 1605 LoC in LLVM framework diff. The custom syscall consists of 419 LoC in the kernel diff.

5.7.1 Spatial Memory Safety

We implemented `llvm-mpx` pass using LLVM [115] to support spatial memory safety on MPX. It instruments at the IR level, protecting heap, address-taken stack and global objects. It follows the same per-pointer bound checking convention used in SoftBound [144] and `gcc-mpx`. Yet, it instruments more instructions than SoftBound: e.g., atomic, vectorization, and invoke instructions. Moreover, `llvm-mpx` models the same set of `libc` functions as `gcc-mpx`: e.g., `malloc`, `memcpy`, and `strcpy`. As the address of the pointers being checked is always taken (e.g., `bndldx(&p)`) in the IR, pointer variables are not promoted to registers, and `llvm-mpx` checks them all. `llvm-mpx` keeps the BT up-to-date, which *BOGO* relies on, while `gcc-mpx` and `icc-mpx` often store bound information in the stack instead of BT.

Optimizations. `llvm-mpx` performs three optimizations during the instrumentation: **(1)** Bound check elimination: if memory access can be statically verified, it elides MPX checks. This is analogous to bound check optimization in the pioneering work of Gupta [87]. `gcc-mpx` also has the same form of optimization. **(2)** Dead bound elimination: The lack of bound checks can make the corresponding bound and the related instructions (e.g., `bndmk/bndldx/bndstx`) dead if they are not "used" by others. it identifies such dead codes by following the use-def chain [140] and eliminates them. **(3)** Bound check consolidation: if it can statically calculate the range of the access in a loop or a vectorized code, it consolidates the checks into one check and pays the overhead only once. This is a very simple form of optimization proposed in Gupta's work [87] and WPBound [222].

Since the above optimizations are safe for spatial memory safety [87, 222], they do not compromise *BOGO*'s temporal memory safety guarantee. For example, optimization (1) only deals with local arrays or globals, not the heap objects that are the target of *BOGO*. Optimization (2) won't trigger if pointers are copied, returned, etc. (i.e., bound is "used").

Bound narrowing. The current prototype does not implement bound narrowing [13]. When a program accesses a specific field of a struct object, the compiler can shrink the bound to that field, rather than the full object, for the fine-grained bounds checking. However, this causes a compatibility issue breaking some SPEC 2006 applications with C idioms due to the resulting false positives [57, 154]. Bound narrowing is optional in `gcc-mpx`, and not supported in many other tools [70, 146, 147, 149, 182] including SoftBound [144, 148].

5.7.2 Temporal Memory Safety

BOGO is built upon LLVM-4.0, glibc-2.23 and linux-4.10. The kernel is modified to support the followings: (1) BTP permission initialization; (2) FullScan optimization (Section 5.4.2); (3) custom `mprotect` avoiding touch unrelated kernel data structures; and (4) signal delivery when a BTP is reclaimed (becomes unavailable) so that it can be removed from HPQ, avoiding a potential segmentation fault during scanning.

5.8 Evaluation

5.8.1 Methodology

We used three sets of benchmarks for evaluation. SPEC CPU 2006 is used in Section 5.8.3 for detailed performance evaluation. The malloc-free benchmark [110, 153] is used in Section 5.8.4 for stress-test. Finally, 9 real-world (multithreaded) applications are tested in Section 5.8.5. The setup is a 4GHz quad-core Intel i7-6700K CPU with 16GB RAM. The performance numbers are the average of 5 runs. Except where otherwise mentioned, all experiments are done with the following configurations: (1) the size of FAQ is 65535 and FullScan is used when it becomes full; (2) the initial size of HPQ is 16 with dynamic adaptive

scheme (Section 5.5.3) enabled; (3) `reference` input is used for SPEC.

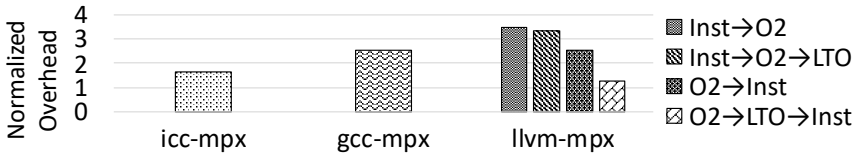


Figure 5.5: MPX compilers overheads: geomean of SPEC 2006.

	Source	Application	Bug manifest
Spatial	BugBench [129]	bc-1.06	storage.c:177 util.c:577
		gzip-1.2.4	gzip.c:828
		man-1.5h1	main.c:977
		ncompress	compress42.c:892
		polymorph-0.4.0	polymorph.c:120,195,198,200,202
	CVE-2004-2167	latex2rtf-1.9.15	definitions.c:155
	CVE-2007-4060	corehttp-0.5.3alpha	http.c:32
	CVE-2011-4971	memcached-1.4.5	memmove()
	CVE-2016-6289	php-7.1Git-2016-06-29	zend_virtual_cwd.c:1243
	CVE-2016-6297	php-7.1Git-2016-06-30	zip_stream.c:289
	CVE-2017-9928	lrzip-0.631	lrzip.c:979
	CVE-2017-9929	lrzip-0.631	lrzip.c:1074
	CVE-2018-5268	OpenCV-3.3.1	grfmt_jpeg2000.cpp:343
	CVE-2018-6187	MuPDF-1.12.0	pdf-write.c:2901
	SPEC 2006	400.perlbench	perlio.c:748, sv.c:4124
		450.soplex	islist.h:287,357 svector.h:351
		464.h264ref	mv-search.c:1016
Temporal	NIST/Juliet [8]	102205 102226 102248 102287 102307 102311 102367 102444 102528 102609 102611 102613 102615 102617 102619 152889 102225 102247 102267 102289 102308 102321 102411 102468 102577 102610 102612 102614 102616 102618 102663 2151	
	CVE-2014-9661	FreeType 2.5.3	ftstream.c:182
	CVE-2015-7801	optipng-0.6.4	opngoptim.c:977
	CVE-2017-10686	nasm-2.14rc0	dereferences of free'd Token obj
	CVE-2017-15642	sox-v14.4.2	formats.c:245

Table 5.2: llvm-mpx and BOGO validation.

Patching Spatial Safety Errors. With `llvm-mpx`, we found the same set of bounds errors

in original SPEC applications as reported in Oleksenko et al.'s work [154] (see their Section 4.4). Thus, we patched [28] them to perform bound-error-free performance evaluation[3].

Instrumentation before or after Optimizations. It is worth noting that `llvm-mpx` pass is applied after standard optimizations including LTO (link-time optimization). That is, apply -O2 for each bitcode file, then perform -O2 LTO to create a single file. `llvm-mpx` is applied at last. The same convention of using all possible optimizations before the instrumentation was adopted in SoftBound [144] and others [145, 148, 190, 221].

We investigated the high runtime overhead of `icc-mpx` and `gcc-mpx`. By scrutinizing the order of applied compiler passes in `gcc-mpx` and `icc-mpx`, we noticed that they first performed MPX instrumentation thus preventing other optimizations. For example, `gcc-mpx`'s instrumentation happens very early in the compiler pass order, i.e., the 12th among 174 passes. And those before the instrumentation are not actually optimization passes. Therefore, all the optimizations can be significantly restricted, e.g., dead code elimination can be suppressed due to the inserted MPX bounds checking code.

Figure 5.5 highlights the impact of the optimize-before-instrument convention on the performance overhead of `llvm-mpx`; each bar represents the average overhead of all SPEC 2006 applications which is normalized to that of baseline with no spatial safety support. By following the convention, our `llvm-mpx` incurs 1.26x slowdown in the 6th bar: O2→LTO→Inst. When we instrument before optimizations like `gcc-mpx` and `icc-mpx`, the overhead is significantly increased, i.e., 3.35x slowdown in the 4th bar: Inst→O2→LTO.

We also found out that Oleksenko et al. [154] did not apply LTO for both `gcc-mpx` and `icc-mpx`. However, it turns out that applying LTO after the instrumentation does not improve the performance significantly. This is confirmed by the small height gap between the

[3]For `soplex`, we manually modified the pointer manipulations that violate standard memory model, and made their bounds checking always succeed. See discussion in [154] Section 4.4.

3rd (Inst→O2) the 4th (Inst→O2→LTO) bars in Figure 5.5. We believe that the same phenomenon will be observed in `gcc-mpx` and `icc-mpx` because LTO is restricted anyway by the inserted MPX bound checks. Thus, we conclude that the reason for the poor performance of `gcc-mpx` and `icc-mpx` is mainly due to their unconformity of optimize-before-instrument convention.

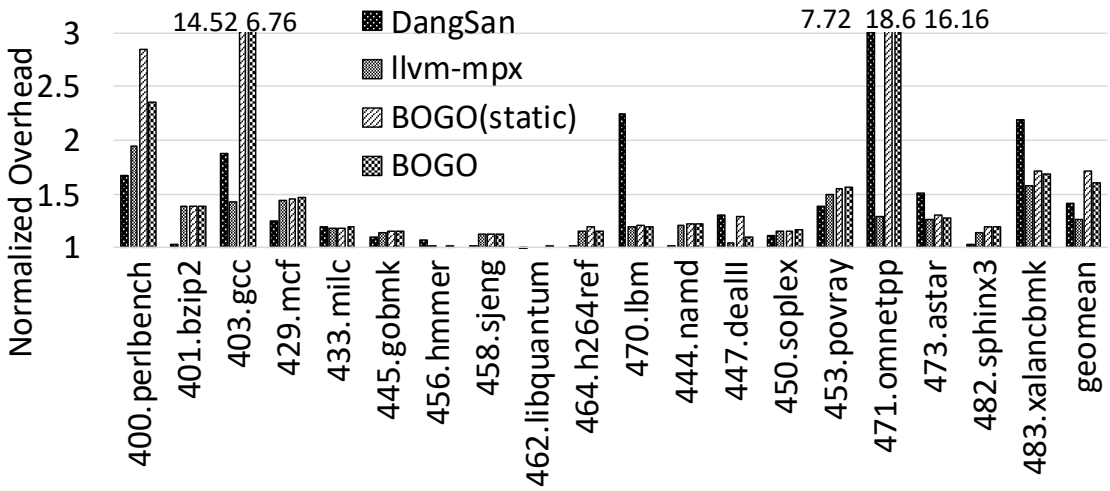


Figure 5.6: Performance Overhead.

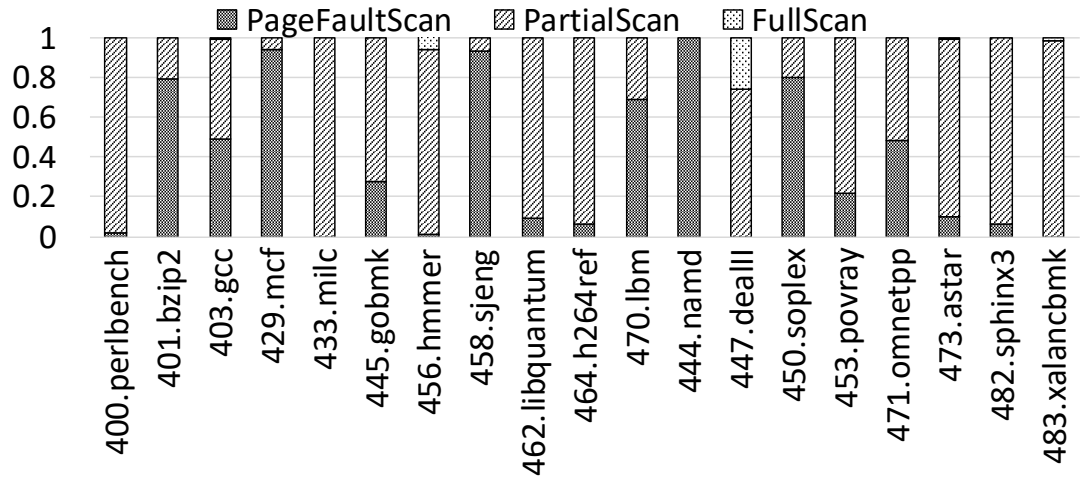


Figure 5.7: *BOGO* Performance Overhead Breakdown.

Application	Partial	Full	Page Fault		Application	Partial	Full	Page Fault
400.perlbench	816926.10	0.49	2875.50		470.lbm	0.03	0	0.01
401.bzip2	0.34	0	0.04		444.namd	4.74	0	0.15
403.gcc	29474.74	0.25	201114.61		447.dealII	770810.54	9.63	0.05
429.mcf	0.03	0	18.30		450.soplex	1432.80	0	641.12
433.milc	18.41	0	0.01		453.povray	17648.83	0	5318.45
445.gobmk	1808.81	0	1386.05		471.omnetpp	73419.43	0.35	214350.09
456.hmmer	4426.50	0	0.28		473.astar	13613.71	0.10	1321.86
458.sjeng	0.01	0	0.01		482.sphinx3	38908.55	0.58	985.62
462.libquantum	0.45	0	0.01		483.xalancbmk	607803.72	0.44	0.25
464.h264ref	514.04	0	599.12					

Table 5.3: Frequency of Partial, Full, and Page Fault Scans (per second). The sum of Partial and Full Scans represents `free` frequency.

5.8.2 Security Evaluation

`llvm-mpx`'s Spatial Memory Safety. *BOGO*'s ability to detect use-after-free hinges on spatial memory safety solution. Thus, it is critical to validate whether `llvm-mpx` is sound. For a fair and accurate comparison, we picked LLVM-based SoftBound [144] as baseline, instead of comparing across different compilers (e.g., `llvm-mpx` vs. `gcc-mpx`). To this end, we collected the number of dynamic bounds checks performed on a subset of tested applications with reference input; the open source version of SoftBound [144] currently works for only 6 SPEC applications, all of which we tested. We confirmed that `llvm-mpx` performs a higher number of bounds checks than SoftBound because the former supports more instructions (Section 5.7.1). Oleksenko et al. [154] also report that `gcc-mpx` leads to a much higher instruction count than `icc-mpx` (∼3x vs. ∼1.5x – see their Figure 10). To further validate `llvm-mpx`, we tested real-world applications with buffer overflow bugs[4]. As in Table 5.2, the first 5 cases are from BugBench [129] followed by 9 CVEs. `llvm-mpx` detected all without false positives. Note that `llvm-mpx` also detected known bugs in SPEC [28].

Based on the above validation steps, we conclude that our `llvm-mpx` implementation is cred-

[4]RIPE [212] is not used as it does not support the 64-bit system.

ible thus being able to serve as a solid basis for *BOGO*'s temporal safety enforcement.

***BOGO*'s Temporal Memory Safety.** We empirically evaluated *BOGO*'s implementation for temporal memory safety. First, we inspected *BOGO*'s detection capability for 32 cases from NIST/Juliet (CWE416, Use-After-Free) [8], as listed in Table 5.2. *BOGO* soundly detected them all. Second, *BOGO* detected all use-after-free vulnerabilities in 4 tested CVEs.

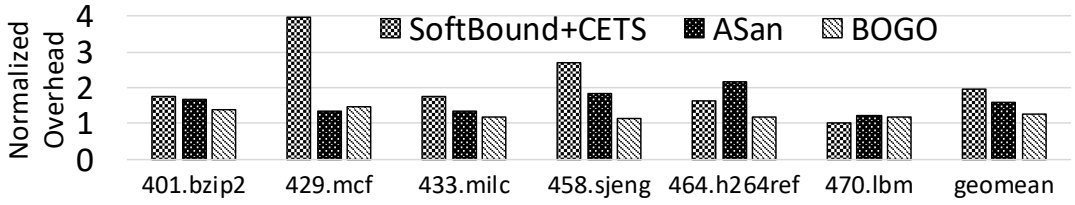


Figure 5.8: Performance Overhead of Full Memory Safety Solutions.

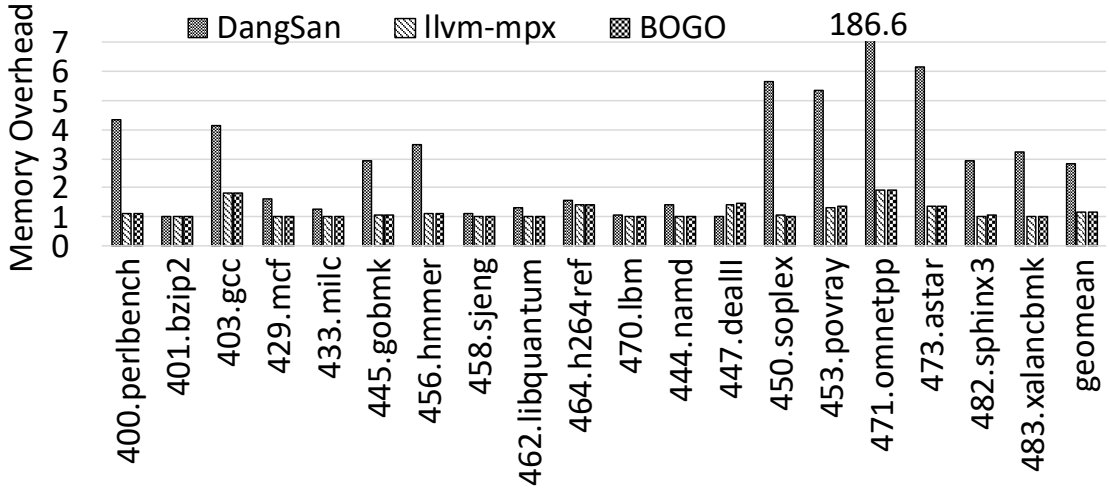


Figure 5.9: Memory Overhead.

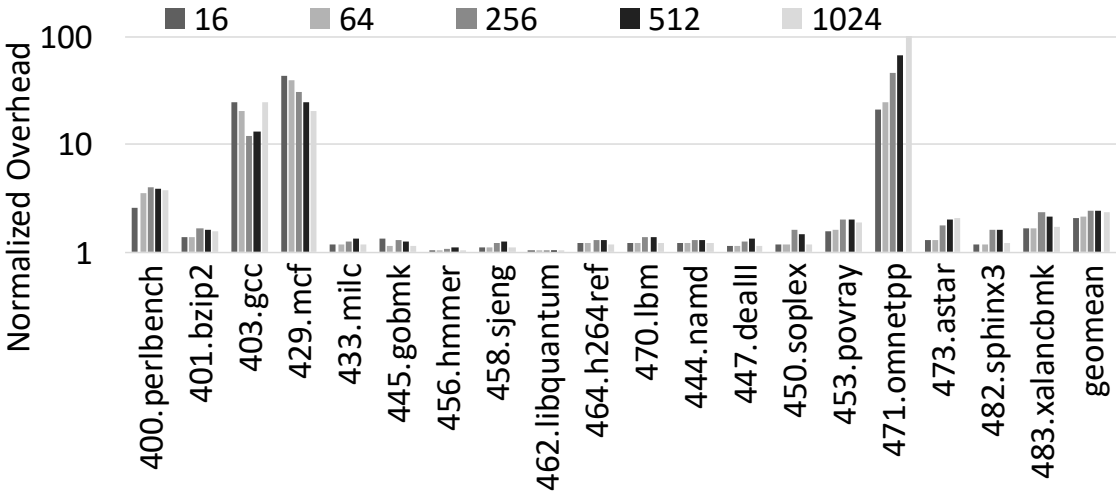


Figure 5.10: Sensitivity study: varying HPQ, fixed-size FAQ.

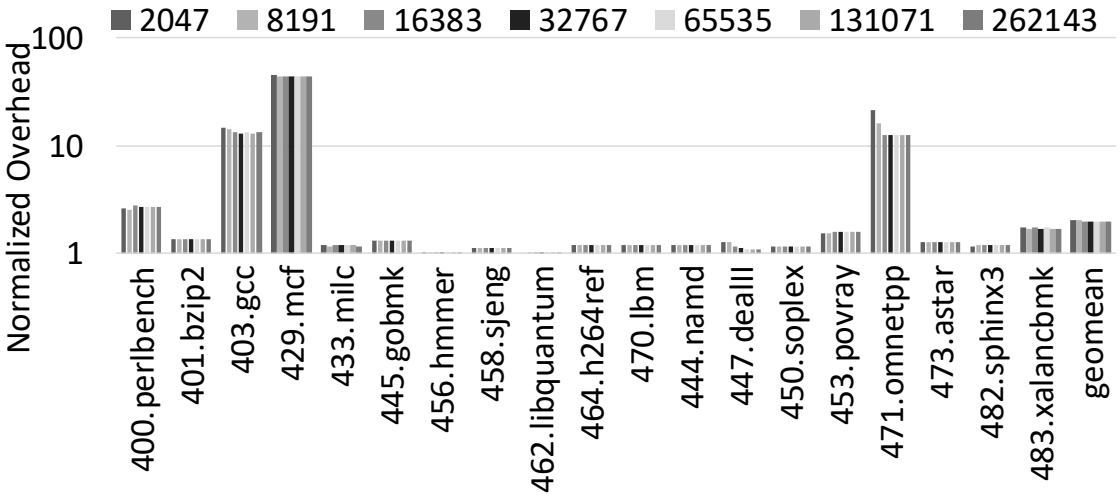


Figure 5.11: Sensitivity study: varying FAQ, fixed-size HPQ.

5.8.3 SPEC CPU 2006 Benchmark

Performance Overhead.

Fig. 5.6 shows the performance overhead normalized to the baseline without memory safety. For each application, there are four bars to compare: DangSan (temporal-only), llvm-mpx (spatial-only), *BOGO* (static), and *BOGO*. The first bar is for DangSan, the state-of-the-art

temporal memory safety only solution. It does not support use-after-return. We evaluated it using the open source version from GitHub [18]. On average, DangSan incurs 1.41x slowdown. The second bar is for `llvm-mpx`. On average, `llvm-mpx` incurs 1.26x slowdown. The next two bars show the performance overhead of *BOGO* (static) and *BOGO* which provide both spatial (via `llvm-mpx`) and temporal memory safety. The third bar, i.e., *BOGO* (static), reports the best per-app result selected by profiling with varying HPQ size (the sensitivity study on HPQ is shown in Section 5.8.3). The last bar, i.e., *BOGO*, shows that its scan cost-based dynamic adaptation of HPQ size (Section 5.5.3) outperforms the best static configuration, i.e., *BOGO* (static). We observed significant improvements for `perlbench`, `gcc` where *BOGO* could dynamically adapt to the phase changes of runtime execution behaviors. Note, the dynamic adaptation scheme can offer similar or better performance without prior profiling or knowledge of the program behavior. On average, *BOGO* incurs 1.6x slowdown for spatial and temporal safety.

We then present detailed performance overhead analysis for *BOGO*. First, Table 5.3 shows the frequencies of PartialScan, FullScan, and PageFaultScan (numbers/sec). Overall, the frequency of FullScan is very small shows the benefit of PartialScan+PageFaultScan. We found that naive FullScan only approach incurs more than 10x slowdown (not shown in Fig. 5.6). Note that the sum of Partial and Full Scan frequencies represents `free` frequency, and it varies significantly across different applications (up to 817K/sec). This implies that the SPEC benchmarks cover a broad spectrum of the deallocation behaviors which affect *BOGO*'s performance overhead. Later, we stress-test *BOGO* with malloc-free intensive benchmarks (Section 5.8.4) and evaluate it with real-world applications (Section 5.8.5).

Fig. 5.7 reports *BOGO*'s performance overhead breakdown of time spent for three scans. Two applications incur relatively high *BOGO* overhead: `gcc` and `omnetpp`. It turns out that `gcc` and `omnetpp` have frequent Partial and Page Fault Scans as shown in Table 5.3. In

general, applications with higher scan frequencies (e.g., `perlbench`, `xalancbmk`) incur higher overhead compared to the others. `gcc` and `omnetpp` also suffer from scanning larger dataset in HPQ and FAQ, According to Fig. 5.7, `gcc` and `omnetpp` spent about 50:50 on PartialScan and PageFaultScan, showing the effectiveness of HPQ dynamic scheme for applications with such high overhead.

Other Full Memory Safety Techniques

Overhead would add up when combining a temporal memory safety solution (e.g. DangSan) with another spatial memory safety solution (e.g., `llvm-mpx`). As shown in Fig. 5.6, DangSan and `llvm-mpx` incur 1.41x and 1.26x slowdown, respectively. When combined, the total runtime overhead would be similar to *BOGO* (1.6x), yet *BOGO* is more memory efficient (Section 5.8.3).

SoftBound+CETS [145, 148] keeps separate per-pointer metadata for spatial memory safety (SoftBound) and temporal memory safety (CETS). Fig. 5.8 highlights the performance of *BOGO* compared to other full memory safety solutions for 6/19 SPEC 2006 applications; the latest open source version of SoftBound+CETS [24] is broken for the remaining 13 applications. For those 6 applications, *BOGO* (1.25x slowdown) significantly outperforms SoftBound+CETS (1.94x); (Their paper reports 1.75x overhead for 9/19 SPEC 2006 and 8/16 SPEC 2000 applications). It seems that the open source version might be less optimized. Although SoftBound+CETS supports use-after-return detection, it is disabled for fair comparison. **AddressSanitizer** [181] maintains per-object metadata. Their paper reports 1.73x slowdown for SPEC 2006, higher than *BOGO* (1.6x). When we run it, with use-after-return disabled, for the same 6 applications in Fig. 5.8, it incurs 1.57x slowdown which is still higher than *BOGO* (1.25x) but lower than the SoftBound+CETS (1.94x). AddressSanitizer quarantines freed memory and defer actual reclamation to support use-after-free detection,

causing memory bloat. To avoid high memory overhead, it does actual free periodically, thereby sacrificing soundness.

Memory Overhead

Figure 5.9 shows the memory usage of DangSan (temporal-only), `llvm-mpx` (spatial-only), and *BOGO*. It is measured by taking the average of resident memory (VmRSS) and normalized to the baseline without memory safety. On average, DangSan incurs 2.84x memory overhead, which is slightly higher than what the paper reports (2.4x) [199], while llvm-mpx and *BOGO* incur 1.16x and 1.17x, respectively. Thus, *BOGO* adds very small memory overhead for full memory safety. When DangSan is combined with MPX for the full safety, the total space overhead would add up. For `omnetpp`, DangSan suffers from 186.62x (134.65x according to the paper [199]) overhead, while *BOGO* incurs only 1.92x overhead compared to llvm-mpx (1.91x). Although DangSan maintains a huge pointer graph for performance, it still incurs 7x slowdown for `omnetpp` as shown in Fig. 5.6.

Sensitivity Study

Dynamic adaptation is disabled in this study. Figures 5.10 and 5.11 show the performance sensitivity studies with respect to the sizes of HPQ and FAQ, respectively. Figure 5.10 shows the result of varying HPQ with the fixed-size (65535) FAQ. As discussed in Section 5.5.2, the size of HPQ affects PartialScan and PageFaultScan in opposing directions. Each application favors different sizes of HPQ. For example, `omnetpp` prefers small HPQ, `mcf` favors larger HPQ, and `gcc` works best on the middle size HPQ. This justifies *BOGO*'s dynamic HPQ size adaptation, and Fig. 5.6 confirms its effectiveness. On the other hand, a bigger FAQ is in general preferable as it reduces the number of FullScans. For example, `omnetpp` in

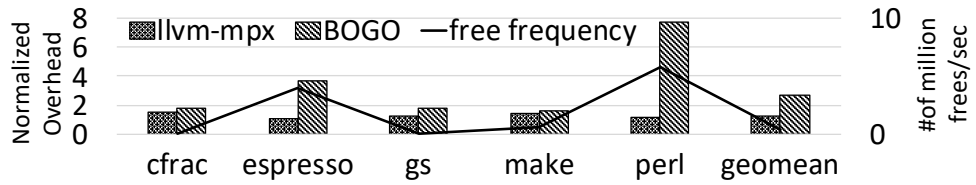


Figure 5.12: Malloc-free benchmark performance. The bar graphs shows the normalized overhead (left y-axis). The line graph shows the free frequency (right y-axis).

Application	LOC	Test method	Free Freq.
aget	1K	download 4GB file, 8 threads	0.45
pfscan	2K	search 4GB file, 8 threads	27
pbzip2	6K	compress 4GB file, 8 threads	5,280
transmission	116K	download file 3.8GB	13,354
memcached	18K	YCSB [61], workload ABCDEF	0.35
cherokee	102K	ab [10], 8 conc. clients, 100K req.	27,509
nginx	166K	ab [10], 8 conc. clients, 100K req.	115,113
apache	270K	ab [10], 8 conc. clients, 100K req.	486
mysql	1,473K	sysbench [29], 8 conc. clients, 100K req.	677,774

Table 5.4: Real-world application test methods. Top four applications are utilities/clients, while the bottom fives are servers.

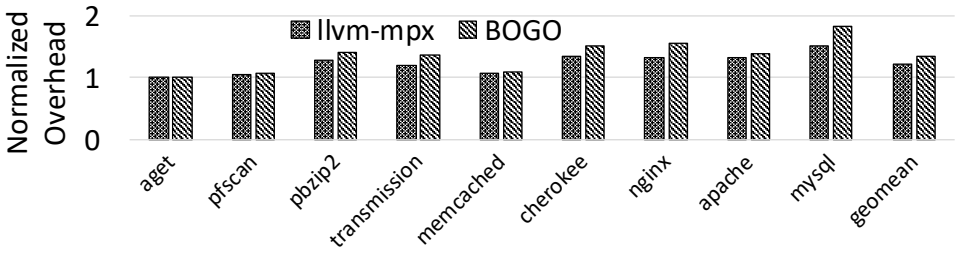


Figure 5.13: Real-world application performance.

Figure 5.11 definitely favors a larger FAQ. We found that the rest applications are not very sensitive to the size of FAQ, though there is some fluctuation. For this reason, *BOGO* does not currently adjust FAQ on the fly.

5.8.4 Malloc/Free Benchmark

Stress-testing *BOGO* with malloc/free intensive applications [110, 153] shows higher runtime overhead (Fig. 5.12) than SPEC: on average, 2.7x slowdown for *BOGO*, and 1.27x for

`llvm-mpx` only. The reason is two-fold: (1) the huge amount of malloc/free (up to 5.8M/s) puts significant pressure on *BOGO*'s page scan mechanism; and (2) the execution time of the applications is very short (less than 2 seconds) even with the largest input, and the majority of the entire execution time is spent allocating/deallocating numerous objects. Thus, *BOGO*'s activity ends up taking a significant portion. However, except for `perl` (5.8M/s) and `espresso` (3.9M/s), the overhead of the remaining applications is under 80%, and the overhead added by *BOGO* upon `llvm-mpx` is only 34%.

5.8.5 Real-World Applications

We evaluated *BOGO* with 9 real-world applications using the test cases listed in Table 5.4. The five servers including `apache` and `mysql` are set up with default configuration. While `nginx` is a single-threaded multi-process server, all the rest 8 are multithreaded applications. As discussed in Section 5.6, *BOGO* does not guarantee soundness for multithreaded applications as with others [144, 145, 154, 199]. Thus, this experiment is just for performance evaluation and compatibility demonstration purposes. We note that all instrumented applications behave correctly. Despite no soundness guarantee, as reported in Section 5.8.2, *BOGO* (llvm-mpx) could detect a buffer overflow bug in `memcached-1.4.5`. Given the workloads, the free frequency varies up to 678K per second. As shown in Fig. 5.13, the runtime overhead of *BOGO* ranges from 1x to 1.83x, with a geomean of 1.34x, which is less than that of more CPU-intensive and malloc/free-frequent SPEC applications.

5.9 Summary

In this chapter, we tackle the space overhead problem of dynamic memory safety bug detectors (**PS2**). We presents *BOGO*, which seamlessly adds temporal memory safety to the spatial memory safety on Intel MPX. *BOGO* scans bound metadata to find dangling pointers, invalidates their bounds, and detects temporal memory safety bugs as spatial safety bugs. This frees *BOGO* from maintaining separate metadata for temporal memory safety, saving both runtime and space overhead (**TS2**). Our evaluation shows that *BOGO* supports full memory safety at comparable runtime overhead and much less memory overhead than other state-of-the-art solutions. *BOGO* is published at [232].

Chapter 6

A Permission Check Analysis Framework for Linux Kernel

So far we have discussed how to improve run-time and space overheads of dynamic bug detectors. In this chapter, we focus on static bug detectors and propose to use domain-specific knowledge to improve the scalability and precision (**PS3**) of static bug detectors altogether. Furthermore, we study permission check bug in the Linux kernel and apply common programming patterns to kernel analysis in order to achieve scalability and precision.

Permission checks play an essential role in operating system security by providing access control to privileged functionalities. However, it is particularly challenging for kernel developers to correctly apply new permission checks and to scalably verify the soundness of existing checks due to the large code base and complexity of the kernel. In fact, Linux kernel contains millions of lines of code with hundreds of permission checks, and even worse its complexity is fast-growing.

In this chapter, we presents PeX, a static Permission check error detector for LinuX, which takes as input a kernel source code and reports any missing, inconsistent, and redundant permission checks. PeX uses KIRIN (Kernel InteRface based Indirect call aNalysis), a novel, precise, and scalable indirect call analysis technique, leveraging the common programming paradigm used in kernel abstraction interfaces. Over the interprocedural control flow graph built by KIRIN, PeX automatically identifies all permission checks and infers the mappings

between permission checks and privileged functions. For each privileged function, PeX examines all possible paths to the function to check if necessary permission checks are correctly enforced before it is called.

We evaluated PeX on the latest stable Linux kernel v4.18.5 for three types of permission checks: Discretionary Access Controls (DAC), Capabilities, and Linux Security Modules (LSM). PeX reported 36 new permission check errors, 14 of which have been confirmed by the kernel developers.

6.1 Introduction

Access control [174] is an essential security enforcement scheme in operating systems. They assign users (or processes) different access rights, called permissions, and enforce that only those who have appropriate permissions can access critical resources (e.g., files, sockets). In the kernel, access control is often implemented in the form of *permission checks* before the use of *privileged functions* accessing the critical resources.

Over the course of its evolution, Linux kernel has employed three different access control models: Discretionary Access Controls (DAC), Capabilities, and Linux Security Modules (LSM). *DAC* distinguishes privileged users (a.k.a., root) from unprivileged ones. The unprivileged users are subject to various permission checks, while the root bypasses them all [16]. Linux kernel v2.2 divided the root privilege into small units and introduced *Capabilities* to allow more fine-grained access control. From kernel v2.6, Linux adopted *LSM* in which various security hooks are defined and placed on critical paths of privileged operations. These security hooks can be instantiated with custom checks, facilitating different security model implementations as in SELinux [187] and AppArmor [11].

Unfortunately, for a new feature or vulnerability found, these access controls have been applied to the Linux kernel code in an ad-hoc manner, leading to *missing*, *inconsistent*, or *redundant* permission checks. Given the ever-growing complexity of the kernel code, it is becoming harder to manually reason about the mapping between permission checks and privileged functions. In reality, kernel developers rely on their own judgment to decide which checks to use, often leading to over-approximation issues. For instance, *Capabilities* were originally introduced to solve the "super" root problem, but it turns out that more than 38% of *Capabilities* indeed check `CAP_SYS_ADMIN`, rendering it yet another root [15].

Even worse, *there is no systematic, sound, and scalable way to examine whether all privileged functions (via all possible paths) are indeed protected by correct permission checks.* The lack of tools for checking the soundness of existing or new permission checks can jeopardize the kernel security putting the privileged functions at risk. For example, DAC, CAP and LSM introduce hundreds of security checks scattered over millions of lines of the kernel code, and it is an open problem to verify if all code paths to a privileged function encounter its corresponding permission check before reaching the function. Given the distributed nature of kernel development and the significant amount of daily updates, chances are that some parts of the code may miss checks on some paths or introduce the inconsistency between checks, weakening the operating system security.

We designed PeX (Chapter 6), a static permission check analysis framework for Linux kernel. Our approach makes it possible to soundly and scalably detect any missing, inconsistent and redundant permission checks in the kernel code. At a high level, it statically explores all possible program paths from user-entry points (e.g., system calls) to privileged functions and detects permission check errors therein. Suppose it finds a path in which a privileged function, say `PF`, is protected (preceded) by a check, say `Chk` in one code. If it is found that any other paths to `PF` bypass `Chk`, then it is a strong indication of a missing check. Similarly,

it can detect inconsistent and redundant permission checks. While conceptually simple, it is very challenging to realize a sound and precise permission check error detection at the scale of Linux kernel.

In particular, there are two daunting challenges that we should address. First, Linux kernel uses indirect calls very frequently, yet its static call graph analysis is notoriously difficult. The latest Linux kernel (v4.18.5) contains 15.8M LOC, 247K functions, and 115K indirect callsites, rendering existing precise solutions (e.g., SVF [189]) unscalable. Only workaround available to date is either to apply the solutions unsoundly (e.g., only on a small code partition as with K-Miner [82]) or to rely on naive imprecise solutions (e.g., type-based analysis). Either way leads to undesirable results, i.e., false negatives (K-Miner) or positives (type-based one).

For a precise and scalable indirect call analysis, we introduce a novel solution called *KIRIN* (Kernel InteRface based Indirect call aNalysis), which leverages kernel abstraction interfaces to enable precise yet scalable indirect call analysis. Our experiment with Linux v4.18.5 shows that KIRIN allows us to detect many previously unknown permission check bugs, while other existing solutions either miss many of them or introduce too many false warnings.

Second, unlike Android which has been designed with the permission-based security model in mind [9], Linux kernel does not document the mapping between a permission check and a privileged function. More importantly, the huge Linux kernel code base makes it practically impossible to review them all manually for the permission check verification.

To tackle this problem, we presents a new technique which takes as input a small set of known permission checks and automatically identifies all other permission checks including their wrappers. Moreover, our dominator analysis [140] automates the process of identifying mappings between permission checks and their potentially privileged functions as well. Our

experiment with Linux kernel v4.18.5 shows that starting from a small set of well-known 3 DAC, 3 Capacities, and 190 LSM checks, our approach automatically (1) identifies 19, 16, and 53 additional checks, respectively, and (2) derives 9243 pairs of permission checks and privileged functions.

This work makes the following contributions:

• **New Techniques**: We proposed and implemented PeX, a static permission check analysis framework for Linux kernel. We also developed new techniques that can perform scalable indirect call analysis and automate the process of identifying permission checks and privileged functions.

• **Practical Impacts**: We analyzed DAC, Capabilities, and LSM permission checks in the latest Linux kernel v4.18.5 using PeX, and discovered 36 new permission check bugs, 14 of which have been confirmed by kernel developers.

• **Community Contributions**: We will release PeX as an open source project, along with the identified mapping between permission checks and privileged functions. This will allow kernel developers to validate their codes with PeX, and to contribute to PeX by refining the mappings with their own domain knowledge.

6.2 Examples of Permission Check Errors

This section illustrates different kinds of permission check errors, found by PeX and confirmed by the Linux kernel developers. We refer to those functions, that validate whether a process (a user or a group) has proper permission to do certain operations, as *permission checks*. Similarly, we define *privileged functions* to be those functions which only a privileged process can access and thus require permission checks.

```
1  int scsi_ioctl(struct scsi_device *sdev, int cmd,
   ↪  void __user *arg)
2  {
3    ...
4    case SCSI_IOCTL_SEND_COMMAND:
5      if (!capable(CAP_SYS_ADMIN) ||
       ↪  !capable(CAP_SYS_RAWIO))
6        return -EACCES;
7      return sg_scsi_ioctl(sdev->request_queue, NULL,
       ↪  0, arg);
8    ...
9  }
```

(a) `sg_scsi_ioctl` (Line 7) is called **with** `CAP_SYS_ADMIN` and `CAP_SYS_RAWIO` capability checks (Line 5). `arg` is user space controllable.

```
1  int scsi_cmd_ioctl(struct request_queue *q, ...,
   ↪  void __user *arg)
2  {
3    ...
4    case SCSI_IOCTL_SEND_COMMAND:
5      ...
6      if (!arg)
7        break;
8      err = sg_scsi_ioctl(q, bd_disk, mode, arg);
9      break;
10   ...
11   return err;
12 }
```

(b) `sg_scsi_ioctl` (Line 8) is called **without** capability checks. `arg` is user space controllable.

```
1  int sg_scsi_ioctl(struct request_queue *q, struct
   ↪  gendisk *disk, fmode_t mode, struct
   ↪  scsi_ioctl_command __user *sic)
2  {
3    ...
4    err = blk_verify_command(req->cmd, mode);
5    ...
6    return err;
7  }
8
9  int blk_verify_command(unsigned char *cmd, fmode_t
   ↪  mode)
10 {
11   ...
12   if (capable(CAP_SYS_RAWIO))
13     return 0;
14   ...
15   return -EPERM;
16 }
```

(c) `sg_scsi_ioctl` calls `blk_verify_command`, which checks `CAP_SYS_RAWIO` capability.

Figure 6.1: Capability check errors discovered by PeX.

6.2.1 Capability Permission Check Errors

Figure 6.1 shows real code snippets of Capability permission check errors in Linux kernel v4.18.5. Figure 6.1a shows the kernel function `scsi_ioctl`, in which `sg_scsi_ioctl` (Line 7) is safeguarded by two Capability checks, `CAP_SYS_ADMIN` and `CAP_SYS_RAWIO` (Line 5). To the contrary, `scsi_cmd_ioctl` in Figure 6.1b calls the same function `sg_scsi_ioctl` (Line 8) without any Capability check. These two functions share three similarities. First, both of them are reachable from the userspace by `ioctl` system call. Second, both call `sg_scsi_ioctl` with a userspace parameter, `void __user *arg`. Last, there is no preceding Capability check on all possible paths to them (though `scsi_ioctl` performs two checks).

The kernel is supposed to sanitize userspace inputs and check permissions to ensure that only users with appropriate permissions can conduct certain privileged operations. As SCSI (Small Computer System Interface) functions manipulate the hardware, they should be protected by Capabilities. At first glance, `scsi_ioctl` seems to be correctly protected (while `scsi_cmd_ioctl` misses two Capability checks).

However, delving into `sg_scsi_ioctl` ends up with a different conclusion. As shown in Figure 6.1c, `sg_scsi_ioctl` calls `blk_verify_command`, which in turn checks `CAP_SYS_RAWIO`. Considering all together, `scsi_ioctl` checks `CAP_SYS_ADMIN` once but `CAP_SYS_RAWIO` "twice", leading to a *redundant* permission check. On the other hand, `scsi_cmd_ioctl` checks only `CAP_SYS_RAWIO`, resulting in a *missing* permission check for `CAP_SYS_ADMIN`. In particular, PeX detects this bug as an *inconsistent* permission check because the two paths disagree with each other, and further investigation shows that one is redundant and the other is missing.

```
1  static int do_readlinkat(int dfd, const char __user
   ↪  *pathname, char __user *buf, int bufsiz)
2  {
3    ...
4    error = security_inode_readlink(path.dentry);
5    if (!error) {
6      touch_atime(&path);
7      error = vfs_readlink(path.dentry, buf, bufsiz);
8    }
9    ...
10 }
```

(a) Kernel LSM usage in system call `readlinkat`. `vfs_readlink` (Line 7) is protected by `security_inode_readlink` (Line 4). Both `pathname` and `buf` (Line 1 and Line 7) are user controllable.

```
1  int ksys_ioctl(unsigned int fd, unsigned int cmd,
   ↪  unsigned long arg)
2  {
3    ...
4    error = security_file_ioctl(f.file, cmd, arg);
5    if (!error)
6      error = do_vfs_ioctl(f.file, fd, cmd, arg);
7    ...
8  }
9
10 int xfs_readlink_by_handle(struct file *parfilp,
   ↪  xfs_fsop_handlereq_t *hreq)
11 {
12   ...
13   error = vfs_readlink(dentry, hreq->ohandle, olen);
14   ...
15 }
```

(b) Kernel LSM usage in system call `ioctl`. It calls `security_file_ioctl` (Line 4) to protect `do_vfs_ioctl` (Line 6). `hreq->ohandle` and `olen` are also user controllable.

Figure 6.2: LSM check errors discovered by PeX.

6.2.2 LSM Permission Check Errors

The example of LSM permission check errors is related to how LSM hooks are instrumented for two different system calls `readlinkat` and `ioctl`.

Figure 6.2a shows the LSM usage in the `readlinkat` system call. On its call path, `vfs_readlink` (Line 7) is protected by the LSM hook `security_inode_readlink` (Line 4) so that a LSM-based MAC mechanism, such as SELinux or AppArmor, can be realized to allow or deny the `vfs_readlink` operation.

Figure 6.2b presents two sub-functions for the system call `ioctl`. Similar to the above case, `ioctl` calls `ksys_ioctl`, which includes its own LSM hook `security_file_ioctl` (Line 4) be-

fore `do_vfs_ioctl` (Line 6). This is proper design, and there is no problem so far. However, it turns out that there is a path from `do_vfs_ioctl` to `xfs_readlink_by_handle` (Line 10), which eventually calls the same privileged function `vfs_readlink` (see Line 7 in Figure 6.2a and Line 13 in Figure 6.2b). While this function is protected by the `security_inode_readlink` LSM hook in `readlinkat`, that is not the case for the path to the function going through `xfs_readlink_by_handle`. The problem is that SELinux maintains separate 'allow' rules for `read` and `ioctl`. With the *missing* LSM `security_inode_readlink` check, a user only with the 'ioctl allow rule' may exploit the `ioctl` system call to trigger the `vfs_readlink` operation, which should only be permitted by the different 'read allow rule'.

The above two Capability and LSM examples show how challenging it is to ensure correct permission checks. There are no tools available for kernel developers to rely on to figure out whether a particular function should be protected by a permission check; and, (if so) which permission checks should be used.

6.3 Challenges

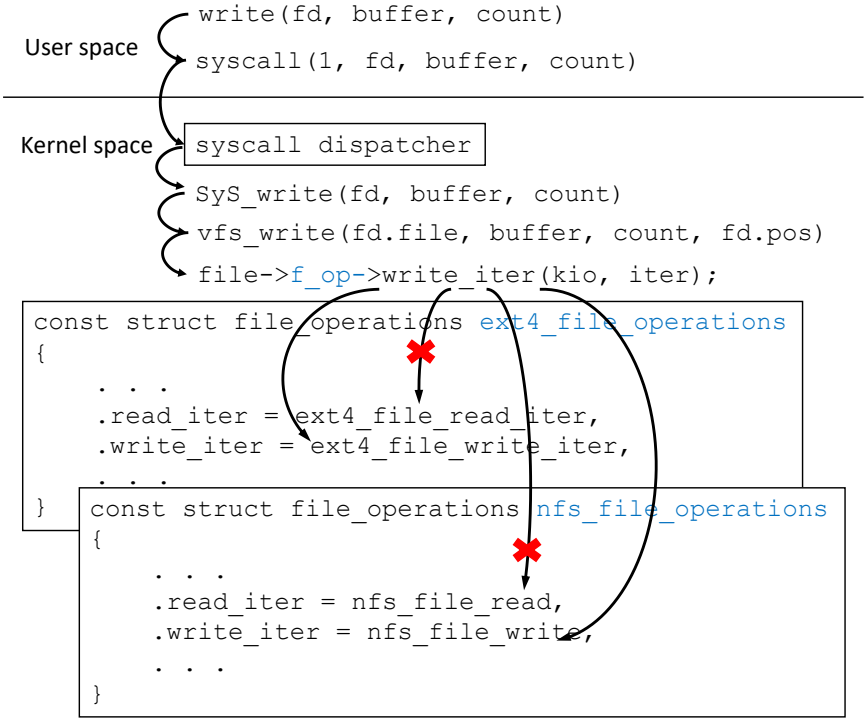
This section discusses two critical challenges in designing static analysis for detecting permission errors in Linux kernel.

6.3.1 Indirect Call Analysis in Kernel

The first challenge lies in the frequent use of indirect calls in Linux kernel and the difficulties in statically analyzing them in a scalable and precise manner. To achieve a modular design, the kernel proposes a diverse set of abstraction layers that specify the common *interfaces* to different concrete implementations. For example, Virtual File System (VFS) [31] abstracts a

```
1  struct file_operations {
2      ...
3      ssize_t (*read_iter) (struct kiocb *, struct
       ↪  iov_iter *);
4      ssize_t (*write_iter) (struct kiocb *, struct
       ↪  iov_iter *);
5      ...
6  }
```

(a) The Virtual File System (VFS) kernel interface.



(b) VFS indirect calls in Linux kernel.

Figure 6.3: Indirect call examples via the VFS kernel interface.

file system, thereby providing a unified and transparent way to access local (e.g., `ext4`) and network (e.g., `nfs`) storage devices. Under this kernel programming paradigm, an abstraction layer defines an interface as a set of indirect function pointers while a concrete module initializes these pointers with its own implementations. For example, as shown in Figure 6.3a, VFS abstracts all file system operations in a *kernel interface* `struct file_operations` that contains a set of function pointers for different file operations. When a file system is initialized, it initializes the VFS interface with the concrete function addresses of its own. For instance, Figure 6.3b shows that `ext4` file system sets the `write_iter` function pointer to

`ext4_file_write_iter`, while `nfs` sets the pointer to `nfs_file_write`.

However, kernel's large code base challenges the resolution of these numerous function pointers within kernel interfaces. For example, the kernel used in our evaluation (v4.18.5) includes 15.8M LOC, 247K functions, and 115K indirect callsites. This huge code base makes existing precise pointer analysis techniques [89, 90, 91, 159, 189] unscalable. In fact, Static Value Flow (SVF) [189], i.e., the state-of-the-art analysis that uses flow- and context-sensitive value flow for high precision, failed to scale to the huge Linux kernel. That is because SVF is essentially a whole program analysis, and its indirect call resolution thus requires tracking all objects such as functions, variables, and so on, making the value flow analysis unscalable to the large-size Linux kernel. In our experiment of running SVF for the kernel on a machine with 256GB memory, SVF was crashed due to an out of memory error[1].

Alternatively, one may opt for a simple "type-based" function pointer analysis, which would scale to Linux kernel. However, the type-based indirect call analysis would suffer from serious imprecision with too many *false* targets, because function pointers in the kernel often share the same type. For example, in Figure 6.3a, two function pointers `read_iter` and `write_iter` share the same function type. Type based pointer analysis will even link `write_iter` to `ext4_file_read_iter` falsely, which may lead to false permission check warnings.

PeX addresses this problem with a new kernel-interface aware indirect call analysis technique, detailed in Section 6.4.

[1]SVF internally uses LLVM SparseVectors to save memory overhead by only storing the set bits. However, it still blows up both the memory and the computation time due to the expensive insert, expand and merge operations.

6.3.2 The Lack of Full Permission Checks, Privileged Functions, and Their Mappings

The second challenge lies in soundly enumerating a set of permission checks and inferring correct mappings between permission checks and privileged functions in Linux kernel.

Though some commonly used permission checks for DAC, Capabilities, and LSM are known (Table 2.1), kernel developers often devise custom permission checks (wrappers) that internally use basic permission checks. Unfortunately, the complete list of such permission checks has never been documented. For example, `ns_capable` is a commonly used permission check for Capabilities, but it calls `ns_capable_common` and `security_capable` in sequence. It is the last `security_capable` that performs the actual capability check. In other words, all the others are "wrappers" of the "basic" permission check `security_capable`. This example shows how hard it is for a permission check analysis tool to keep up with all permission checks.

To make matters worse, Linux kernel has no explicit documentation that specifies which privileged function should be protected by which permission checks. This is different from Android [9], which has been designed with the permission-based security model in mind from the beginning. Take the Android `LocationManager` class as an example; for the `getLastKnownLocation` method, the API document states explicitly that permission `ACCESS_COARSE_LOCATION` or `ACCESS_FINE_LOCATION` is required [22].

Unfortunately, existing *static* permission error checking techniques are not readily applicable in order to address these problems. Automated LSM hook verification [193] works only with clearly defined LSM hooks, which would miss many wrappers in the kernel setting. Many other tools require heavy manual efforts such as user-provided security rules [80, 234], authorization constraints [142], annotation on sensitive objects [81]. These manual processes

are particularly error-prone when applied to huge Linux code base. Alternatively, some works such as [74, 141] rely on *dynamic* analysis. However, such run-time approaches may significantly limit the code coverage being analyzed, thereby missing real bugs.

Moreover, all of above existing works cannot detect permission checks soundly. Their inability to recognize permission checks or wrappers leads to missing privileged functions or false warnings for those that are indeed protected by wrappers. Since the huge Linux kernel code base makes it practically impossible to review them all manually, reasoning about the mapping is considered to be a daunting challenge.

In light of this, PeX presents a novel static analysis technique that takes as input a small set of known permission checks to identify their basic permission checks and leverages them as a basis for finding other permission check wrappers (Section 6.5.2). In addition, PeX proposes a dominator analysis based solution to automatically infer the mappings between permission checks and privileged functions (Section 6.5.3).

6.4 KIRIN Indirect Call Analysis

PeX proposes a precise and scalable indirect call analysis technique, called KIRIN (Kernel InteRface based Indirect call aNalysis), on top of the LLVM [115] framework. KIRIN is inspired by two key observations: (1) almost all (95%) indirect calls in the Linux kernel are originated from kernel interfaces (Section 6.3.1) and (2) the type of a kernel interface is preserved both at its initialization site (where a function pointer is defined) and at the indirect callsite (where a function pointer is used) in LLVM IR. For example in Fig. 6.3b, the kernel interface object `ext4_file_operations` of the type `struct file_operations` is statically initialized where `ext4_file_write_iter` is assigned to the field of `write_iter`. For the indirect call site `file`→`f_op`→`write_iter`, one can identify that `f_op` is of the type `struct`

`file_operations` and infer that `ext4_file_write_iter` is one of potential call targets. Based on this observation, PeX first collects indirect call targets at kernel interface initialization sites (Section 6.4.1) and then resolves them at indirect callsites (Section 6.4.2).

6.4.1 Indirect Call Target Collection

In Linux kernel, a kernel interface is often defined in a `C struct` comprised of function pointers (Section 6.3.1): e.g., `struct file_operations` in Fig. 6.3a. Many kernel interfaces (`C structs`) are *statically* allocated and initialized as with `ext4_file_operations` and `nfs_file_operations` in Fig. 6.3b. Some interfaces may be *dynamically* allocated and initialized at run time for reconfiguration.

For the former, KIRIN scans all Linux kernel code linearly to find all statically allocated and initialized `struct` objects with function pointer fields. Then, for each `struct` object, KIRIN keep tracks of which function address is assigned to which function pointers field using an offset as a key for the field. For instance, Fig. 6.4a shows the LLVM IR of statically initialized `ext4_file_operations`. KIRIN finds that the kernel interface type is `struct file_operations` (Line 1), and `ext4_file_write_iter` is assigned to the 5th field `write_iter` (Line 7). Therefore, KIRIN figures out that `write_iter` may point to `ext4_file_write_iter`, not `ext4_file_read_iter` (even though they have the same function type).

For the rest dynamically initialized kernel interfaces, KIRIN performs a data flow analysis to collect any assignment of a function address to the function pointer inside a kernel interface. KIRIN's field-sensitive analysis allows the collected targets to be associated with the individual field of a kernel interface.

```
1 @ext4_file_operations = dso_local local_unnamed_addr
  ↪ constant %struct.file_operations {
2 %struct.module* null,
3 i64 (%struct.file*, i64, i32)* @ext4_llseek,
4 i64 (%struct.file*, i8*, i64, i64*)* null,
5 i64 (%struct.file*, i8*, i64, i64*)* null,
6 i64 (%struct.kiocb*, %struct.iov_iter*)*
  ↪ @ext4_file_read_iter,
7 i64 (%struct.kiocb*, %struct.iov_iter*)*
  ↪ @ext4_file_write_iter,
```

(a) LLVM IR of `ext4_file_operations` initialization.

```
1 %25 = load %struct.file_operations*,
  ↪ %struct.file_operations** %f_op, align 8
2 %write_iter.i.i = getelementptr inbounds
  ↪ %struct.file_operations,
  ↪ %struct.file_operations* %25, i64 0, i32 5
3 %26 = load i64 (%struct.kiocb*, %struct.iov_iter*)*,
  ↪ i64 (%struct.kiocb*, %struct.iov_iter*)**
  ↪ %write_iter.i.i, align 8
4 %call.i.i = call i64 %26(%struct.kiocb* nonnull
  ↪ %kiocb.i, %struct.iov_iter* nonnull %iter.i) #10
```

(b) LLVM IR of callsite `file→f_op→write_iter` in `vfs_write`.

Figure 6.4: Indirect callsite resolution for `vfs_write`.

6.4.2 Indirect Callsite Resolution

KIRIN stores the result of the above first pass in a key-value map data structure in which the key is a pair of kernel interface type and an offset (a field), and the value is a set of call targets. At each indirect callsite, KIRIN retrieves the type of a kernel interface and the offset from LLVM IR, looks up the map using them as a key, and figures out the matched call targets. For example, Fig. 6.4b shows the LLVM IR snippet in which an indirect call `file→f_op→write_iter` is made inside of `vfs_write`. When an indirect call is made (Line 4), KIRIN finds that the kernel interface type is `struct file_operations` (Line 1) and the offset is 5 (Line 2). In this way, KIRIN reports that `ext4_file_write_iter` (assigned at Line 7 in Fig. 6.4a) is one of potential call targets that are indirectly called by dereferencing `write_iter`.

When applying KIRIN to Linux kernel, we found in certain callsites, the kernel interface

```
1  struct usb_driver* driver =
   ↪  container_of(intf->dev.driver, struct
   ↪  usb_driver, drvwrap.driver);
2  retval = driver->unlocked_ioctl(intf,
   ↪  ctl->ioctl_code, buf);
```

(a) C code of a `container_of` usage, followed by an indirect call.

```
1  #define container_of(ptr, type, member) ({                \
2      void *__mptr = (void *)(ptr);                         \
3      ((type *)(__mptr - offsetof(type, member))); })
4  %unlocked_ioctl = getelementptr inbounds i8*, i8**
   ↪  %add.ptr76, i64 3
```

(b) Original `container_of` and the LLVM IR for the callsite.

```
1  #define container_of(ptr, type, member) ({                \
2      type* __res;                                          \
3      void* __mptr = ((void *)((void*)(ptr) -               \
       ↪  offsetof(type, member)));                          \
4      memcpy(&__res, &__mptr, sizeof(void*));               \
5      (__res);})
6  %unlocked_ioctl = getelementptr inbounds
   ↪  %struct.usb_driver, %struct.usb_driver* %20, i64
   ↪  0, i32 3
```

(c) Modified `container_of` and the LLVM IR for the callsite.

Figure 6.5: Fixing `container_of` missing struct type problem.

type is not presented in the LLVM IR, making their resolution impossible. For example, the macro `container_of` is commonly used in order to get the starting address of a `struct` object by using a pointer to its own member field. Fig. 6.5a shows an example of using `container_of` (Line 1). It calculates the starting address of `usb_driver` through its own member `drvwrap.driver`. Based on the address, the code at Line 2 makes an indirect call by using a function pointer `unlocked_ioctl` that is another member of the `struct usb_driver` object.

Fig. 6.5b shows the original macro `container_of` (Lines 1-3) and resulting LLVM IR (Line 4). The problem of this macro is that it involves a pointer manipulation, the LLVM IR of which voids the `struct` type information, i.e., the second argument of the macro. To solve this problem, KIRIN redefines `container_of` in a way that the `struct` type is preserved in the LLVM IR (on which KIRIN works), as in Fig. 6.5c (Lines 1-5). This adds back the kernel interface type `struct.usb_driver` in the LLVM IR (Line 6), thereby enabling KIRIN to infer

the correct type of `driver` and resolve the targets for `unlocked_ioctl`.

Our experiment (Section 6.6.2) shows that KIRIN resolves 92% of total indirect callsites for `allyesconfig`. PeX constructs a more sound (less missing edges) and precise (less false edges) call graph than other existing workarounds (e.g., [82]).

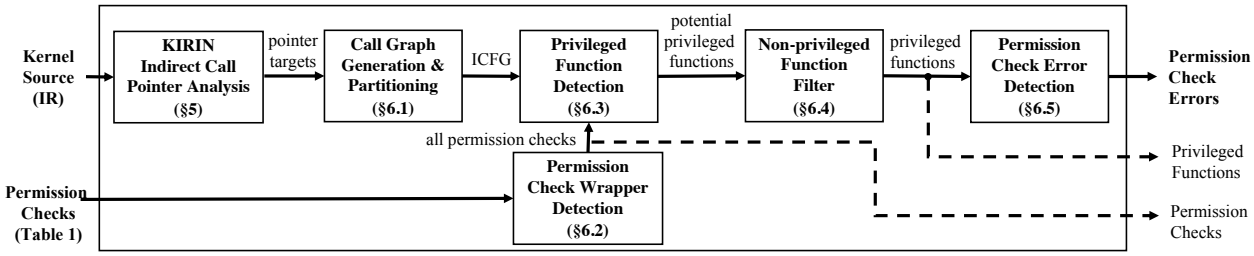


Figure 6.6: PeX static analysis architecture. PeX takes as input kernel source code and permission checks, and reports as output permission check errors. PeX also produces mappings between identified permission checks and privileged functions as output.

6.5 Design of PeX

Fig. 6.6 shows the architecture of PeX. It takes as input kernel source code (in the LLVM bitcode format) and common permission checks (Table 2.1), analyzes and reports all detected permission check errors, including missing, inconsistent, and redundant permission checks. In addition, PeX produces the mapping of permission checks and privileged functions, which has not been formally documented.

At a high-level, PeX first resolves indirect calls with our new technique called KIRIN (Section 6.4). Next, PeX builds an augmented call graph—in which indirect callsites are connected to possible targets—and cuts out only the portion reachable from user space (Section 6.5.1). Based on the partitioned call graph, PeX then generates the interprocedural control flow graph (ICFG) where each callsite is connected to the entry and the exit of the callee [73]. Then, starting from a small set of (user-provided) permission checks, PeX auto-

matically detects their wrappers (Section 6.5.2). After that, for a given permission check, PeX identifies its potentially privileged functions on top of the ICFG (Section 6.5.3), followed by a heuristic-based filter to prune obviously non-privileged functions (Section 6.5.4). Finally, for each privileged function, PeX examines all user space reachable paths to it to detect any permission checks error on the paths (Section 6.5.5). The following section describes these steps in detail.

6.5.1 Call Graph Generation and Partition

PeX generates the call graph leveraging the result of KIRIN (Section 6.4), and then partitions it into two groups.

User Space Reachable Functions: Starting from functions with the common prefix `SyS_` (indicating system call entry points), PeX traverses the call graph, marks all visited functions, and treats them as user space reachable functions. The user reachable functions in this partition are investigated for possible permission check errors.

Kernel Initialization Functions:

Functions that are used only during booting are collected to detect redundant checks. The Linux kernel boots from the `start_kernel` function, and calls a list of functions with the common prefix `__init`. PeX performs multiple call graph traversals starting from `start_kernel` and each of the `__init` functions to collect them.

Other functions such as IRQ handlers and kernel thread functions are not used in later analysis since they cannot be directly called from user space. The partitioned call graph serves as a basis for building an interprocedural control flow graph (ICFG) [140] used in the inference of the mapping between permission checks and privileged functions (Section 6.5.3).

6.5.2 Permission Check Wrapper Detection

Sound and precise detection of permission check errors requires a complete list of permission checks, but they are not readily available (Section 6.3.2). One may name some commonly used permission checks, as in Table 2.1. However, they are often the wrapper of basic permission checks, which actually perform the low-level access control, and even worse there could be other wrappers of the wrapper.

PeX solves this by automating the process of identifying all permission checks including wrappers. PeX takes an incomplete list of user-provided permission checks as input. Starting from them, PeX detects basic permission checks, by performing the *forward call graph slicing* [114, 167, 196] over the augmented call graph. For a given permission check function, PeX searches all call instructions inside the function for the one that passes an argument of the function to the callee. In other words, PeX identifies the callees of the permission check function which take its *actual* parameter as their own *formal* parameter. Similarly, PeX then conducts *backward call graph slicing* [114, 167, 196] from these basic permission checks to detect the list of their wrappers. PeX refers to only those callers that pass permission parameters as wrappers, excluding other callers just using the permission checks.

Fig. 6.7 shows an example of the permission check wrapper detection. Given a known permission check `ns_capable` (Lines 10-13), PeX first finds `security_capable` (Line 4) as a basic permission check, and then based on it, PeX detects another permission check wrapper `has_ns_capability` (Lines 14-20). Note that the parameter `cap` is passed from both the parents `ns_capable_common` and `has_ns_capability` to the child `security_capable`; and the result of `security_capable` is returned to them. Our evaluation (Section 6.6.3) shows that based on 196 permission checks in Table 2.1, PeX detects 88 wrappers.

```
1  static bool ns_capable_common(struct user_namespace
   ↪  *ns, int cap, bool audit)
2  {
3      ....
4      capable = audit ?
       ↪  security_capable(current_cred(), ns, cap) :
5          security_capable_noaudit(current_cred(), ns,
           ↪  cap);
6      if (capable == 0)
7          return true;
8      return false;
9  }
10 bool ns_capable(struct user_namespace *ns, int cap)
11 {
12     return ns_capable_common(ns, cap, true);
13 }
14 bool has_ns_capability(struct task_struct *t,
15                        struct user_namespace *ns, int cap)
16 {
17     ...
18     ret = security_capable(__task_cred(t), ns, cap);
19     ...
20 }
```

Figure 6.7: Permission check wrapper examples.

6.5.3 Privileged Function Detection

It is important to understand the mappings between permission checks and privileged functions for effective detection of any permission check errors therein. However, the lack of clear mapping in Linux kernel complicates the detection of permission check errors (Section 6.3.2).

To address this problem, PeX leverages an interprocedural *dominator* analysis [140] that can automatically identify the privileged functions protected by a given permission check. PeX conservatively treats all targets (callees) of those call instructions, that are dominated by each permission check (Section 6.5.2) on top of the ICFG (Section 6.5.1), as its *potential* privileged functions. The rationale behind the dominator analysis is based on the following observation: since there is no single path that allows the dominated call instruction to be reached without visiting the dominator (i.e., the permission check), the callee is likely to be the one that should be protected by the check on all paths [2].

Algorithm 1 shows how PeX uses the dominator analysis to find potential privileged functions

[2]This does not necessarily mean that the permission check dominates all call instructions of ICFG which invoke the resulting privileged function. As long as some call instructions are dominated by the check, their callees are treated as privileged functions.

Algorithm 1 Privileged Function Detection

INPUT:
pcfuncs - all permission checking functions
OUTPUT:
pvfuncs - privileged functions
1: **procedure** Privileged Function Detection
2: **for** $f \leftarrow pcfuncs$ **do**
3: **for** $u \leftarrow User(f)$ **do**
4: $CallInst \leftarrow CallInstDominatedBy(u)$ ▷ Inter-procedural analysis, for full program path
5: $callee \leftarrow getCallee(CallInst)$
6: $pvfuncs.insert(callee)$
7: **end for**
8: **end for**
9: **return** *pvfuncs*
10: **end procedure**

`pvfuncs` for a given list of permission check functions `pcfuncs`. For each permission check function `f` (Line 2), PeX finds all users of `f`, i.e., the callsite invoking `f` (Line 3). For each user (callsite) `u`, PeX performs interprocedural dominator analysis over the ICFG to find all dominated call instructions (Line 4). All their callees are then added to `pvfuncs` (Lines 5-6).

Note that the call graph generated by KIRIN (Section 6.4) has resolved most of the indirect calls, which allows PeX to perform—on top of the resulting ICFG—more sound privileged function detection. For example, our experiment (Section 6.6.3) shows that KIRIN can identify `ecryptfs_setxattr` (reachable via indirect calls over the ICFG) as a privileged function and detect its missing permission check bug (Table 6.5, LSM-17). Note that if some other unsound workaround such as [82] had been used, this bug could not have been detected.

6.5.4 Non-privileged Function Filter

The conservative approach in Section 6.5.3 may lead to too many potential privileged functions. In this step, PeX applies heuristic-based filters to prune out false privileged functions. In the current prototype, the filter contains a set of kernel library functions which are not privileged functions, e.g., `kmalloc`, `strcmp`, `kstrtoint`. Though PeX is currently designed to avoid false negatives (and thus leverages a small set of library filters only), one can add more

Algorithm 2 Permission Check Error Detection

```
INPUT:
    pc − pv - permission check function to privileged function mapping
    pcfuncs - all permission check functions
    kinitfuncs - kernel init functions
1:  procedure Permission Check Error Detection
2:      for pair ← pc − pv do
3:          pvfuncs ← pair.pv                                   ▷ privileged functions
4:          pcfunc ← pair.pc                                    ▷ permission check functions
5:          for f ← pvfuncs do
6:              allpath ← getAllPathUseFunc(f)   ▷ get all user reachable paths that call the privileged function f
7:              for p ← allpath do
8:                  pvcall ← PrivilegeFunctionCallInPath(p)
9:                  if pvcall not Preceded by pcfunc then
10:                     if pvcall not Preceded by any pcfuncs then
11:                         report(p)                           ▷ Report missing checks
12:                     else
13:                         report(p)                           ▷ Report inconsistent check
14:                     end if
15:                 else if pvcall Preceded by multiple same pcfunc then
16:                     report(p)                               ▷ Report redundant checks
17:                 end if
18:             end for
19:         end for
20:     end for
21:     for f ← kinitfuncs do
22:         if f uses any pcfuncs then
23:             report(f)                   ▷ Report unnecessary checks during kernel boot
24:         end if
25:     end for
26: end procedure
```

aggressive filters to purge more false privileged functions. With releasing PeX, we expect a good opportunity for the kernel development community to contribute to the design of non-privileged function filters where domain knowledge is helpful.

6.5.5 Permission Check Error Detection

This last step is validating the use of privileged functions to detect any potential permission check errors. For a given mapping between a permission check and a privileged function, PeX performs a backward traversal of the ICFG, starting from the privileged functions with the corresponding permission check in mind. Note that PeX validates every possible path to each privileged function of interest.

Algorithm 2 shows PeX's permission check error detection algorithm. Recall that PeX treats

user reachable kernel functions and kernel initialization functions separately and detects different forms of errors (Section 6.5.1). Lines 2-12 shows how PeX detects missing, redundant, and inconsistent checks in user reachable kernel functions. For each privileged function `f` (Line 5) in a mapping, PeX finds all possible paths `allpath` from user entry points to that privileged function `f` over the ICFG (Line 6). Line 7-18 checks each path `p` for the preceding permission check function, the lack of which should be reported as a bug. If the call to the privileged function (`pvcall`) is not preceded by the corresponding permission check function (`pcfunc`) and any other check functions (those in `pcfuncs`) over a given path `p`, then PeX reports a missing check (Lines 6-7). And if `pvcall` is preceded not by the corresponding check (`pcfunc`) but other check in `pcfuncs`, PeX reports an inconsistent check. Finally, if PeX discovers that `pvcall` is indeed preceded by `pcfunc` checks but multiple times, then it reports a redundant check (Lines 15-17). Besides, Lines 21-25 shows how PeX detects redundant checks in kernel initialization functions. As `kinitfuncs` includes a conservative list of functions that can only be executed during booting (thus obviating the need of any checks), all detected permission checks are marked as redundant (Lines 22-24).

6.6 Implementation and Evaluation

PeX was implemented using LLVM [115]/Clang-6.0. It contains about 7K lines of C/C++ code. Clang was modified to preserve the kernel interface type at allocation/initialization sites by using an *identified struct* type instead of using unnamed *literal struct* type. We also automated the generation of the single-file whole vmlinux LLVM bitcode `vmlinux.bc` using `wllvm` [33]. This avoids building each kernel module separately or changing kernel build infrastructures, as observed in prior kernel static analysis works [82, 207]. We evaluated PeX on the latest stable Linux kernel v4.18.5. In summary, KIRIN resolves 86%–92% of indirect

Table 6.1: Input Statistics for Kernel v4.18.5.

	defconfig	**allyesconfig**
# of yes(=y) config	1284	9939
# of compiled LOC	2,414,772	15,881,692
vmlinux size	481 MB	3.8 GB
vmlinux.bc size	387 MB	3.3 GB
# of total functions	42,264	247,465
# of syscall entries	857	1,027
# of init functions	1,570	9,301
# of indirect callsites (ICS)	20,338	115,537

callsites depending on its compilation configurations. PeX reported 36 permission check errors warnings to the Linux community, 14 of which have been confirmed as real bugs.

6.6.1 Evaluation Methodology

We evaluated PeX with two different kernel configurations: (1) `defconfig`, the (commonly-used) default configuration, and (2) `allyesconfig` with all non-conflict configuration options enabled. The use of `allyesconfig` not only stress-tests PeX (including KIRIN) with a larger code base than `defconfig`, but also covers the majority of kernel code, allowing PeX to detect more bugs.

In addition, we used 3 DAC, 3 Capabilities, and 190 LSM permission checks(Table 2.1) as input permission checks, from which PeX finds other wrappers. For the non-privileged function filter, we collected 1827 library functions from `lib` directory in the kernel source code. All experiments were carried out on a machine running Ubuntu 16.04 with two Intel Xeon E5-2650 2.20GHz CPU and 256GB DRAM.

Table 6.2: Indirect Call Pointer Analysis.

	defconfig			allyesconfig		
	KIRIN	**TYPE**	**KM**	**KIRIN**	**TYPE**	**KM**
% of ICS resolved	86	100	1	92	100	na
# of avg target	3.6	10K	3.6	6.2	81K	na
analysis time (min)	1	1	9,869	6.6	1	na

6.6.2 Evaluation of KIRIN

We compared the effectiveness and efficiency of KIRIN with type-based approach and SVF-based K-Miner approach.

K-Miner [82] works around the scalability problem in SVF by analyzing the kernel on a per system call basis, rather than taking the entire kernel code for analysis. K-Miner generates a (small-size) partition of kernel code which can be reached from a given system call, and (unsoundly) applies SVF for that partition. For comparison, we took K-Miner's implementation from the github [19] and added the logic to count the number of resolved indirect callsites and the average number of targets per callsite. As K-Miner was originally built on LLVM/Clang-3.8.1, we recompiled the same kernel v4.18.5 using `wllvm` with the same kernel configurations.

Table 6.2 summaries evaluation results of KIRIN, comparing it to the type-based approach and K-Miner approach in terms of the percentage of indirect callsite (ICS) resolved, the average number of targets per ICS, and the total analysis time.

Resolution Rate

For K-Miner, we observe somewhat surprising results: it resolves only 1% of all indirect callsites. After further investigation, we noticed that SVF runs on each partition whose code base is smaller than the whole kernel, its analysis scope is significantly limited and unable to resolve function pointers in other partitions, leading to the poor resolution rate.

Besides, we found out that K-Miner does not work for `allyesconfig` which contains a much larger code base than `defconfig`. Note that K-Miner evaluated its approach only for `defconfig` in the original paper [82]. The K-Miner approach turns out to be not scalable to handle `allyesconfig` which ends up encountering out of memory error even for analyzing a single system call.

Resolved Average Targets

For KIRIN, the number of average indirect call targets per resolved indirect callsite is much smaller than that of the type-based approach as shown in the second row of Table 6.2. The reason is that the type-based approach classifies all functions (not only address-taken functions) into different sets based on the function type. This implies that all functions in the set are regarded as possible call targets of that function pointer. For example, as shown in Fig. 6.3a, two functions `ext4_file_read_iter` and `ext4_file_write_iter` share the same type. As a result, the type-based approach incorrectly identifies both functions as possible call targets of the function pointer `f_ops→write_iter`.

Analysis Time

The total analysis times of each ICS resolution approach are shown in the last row of Table 6.2. As expected, the type-based approach is the fastest, finishing analysis in 1 minute for both configurations. KIRIN runs slower than the type-based approach. However, the analysis time of KIRIN (≈1 minute) is comparable to that of the type-based approach for `defconfig`, while KIRIN takes 6.6 minutes for `allyesconfig`.

For a fair comparison with K-Miner, care must be taken when we collect its indirect call analysis time. For a given system call, we measured K-Miner's running time from the

Table 6.3: PeX Results.

	defconfig			allyesconfig		
	DAC	**CAP**	**LSM**	**DAC**	**CAP**	**LSM**
# of input checks	3	3	190	3	3	190
# of detected wrappers	11	13	34	19	16	53
# of priv func detected	174	869	2030	631	3770	10915
# of priv func after filter	116	582	1635	537	3245	10260
# of warnings grouped by priv func	72	210	853	221	850	1017
total time (min)	6	8	11	83	247	169

Table 6.4: Comparison of PeX warnings when used with different indirect call analyses.

	defconfig				allyesconfig			
	DAC	**CAP**	**LSM**	Bugs	**DAC**	**CAP**	**LSM**	Bugs
KIRIN	72	210	853	21	221	850	1017	36
TYPE	218	348	1319	21	164	964	4364	19 (PeX Timeout)
KM	54	196	241	6	na	na	na	na (SVF Timeout)

beginning until it produces the SVF point-to result, which does not include the later bug detection time. To obtain the total analysis time of K-Miner, we summed up the running times of all system calls. The result shows that SVF based K-Miner takes about 9,869 minutes to finish analyzing all system calls of `defconfig`, which is much slower than KIRIN's.

6.6.3 PeX Result

Table 6.3 summarizes PeX's intermediate program analyses. As `allyesconfig` subsumes `defconfig` in static analysis, we focus on discussing `allyesconfig` results here. Overall, PeX finishes all analyses within a few hours and reports about two thousand groups of warnings, which are classified by privileged functions. One may implement a multi-threaded version of PeX to further reduce the analysis time.

Given the small number of input DAC, CAP, and LSM permission checks (3, 3, and 190 each), PeX's permission check detection (Section 6.5.2) was able to identify 19, 16 and 53 permission check wrappers. For example, PeX detects wrappers such as `nfs_permission` and `may_open`

for DAC; `sk_net_capable` and `netlink_capable` for Capabilities; and `key_task_permission` and `__ptrace_may_access` for LSM.

Table 6.3 also shows the number of potentially privileged functions detected by PeX (Section 6.5.3) and the number of remaining privileged functions after kernel library filtering (Section 6.5.4). We found that there are typically 1-to-1 or 2-to-1 mapping between permission checks and privileged functions. Overall, PeX reports 221, 850, and 1017 warnings (grouped by privileged functions) for DAC, CAP, and LSM, respectively.

Table 6.5 shows the list of 36 bugs we reported, 14 of which have been confirmed by Linux kernel developers. Kernel developers ignored some bugs and decided not to make changes because they believe that the bugs are not exploitable. We discuss them in detail in Section 6.6.5.

Comparison. To highlight the effectiveness of KIRIN, we repeated the end-to-end PeX analysis using type-based (PeX+TYPE) and K-Miner-style (PeX+KM) indirect call analyses. Table 6.4 shows the resulting number of warnings and detected bugs when the 36 bugs—shown in Table 6.5—are used as an oracle for false negative comparison.

For `allyesconfig`, PeX+TYPE and PeX+KM could not complete the analysis within the 12-hour experiment limit. PeX+TYPE generated too many (false) edges in ICFG and suffered from path explosion during the last phase of PeX analysis; only 19 bugs were reported before the timeout. In the mean time, PeX+KM timed out on an earlier pointer analysis phase, thereby failing to report any bug.

When `defconfig` is used for comparison, PeX+TYPE and PeX+KM were able to complete the analysis. In this setting, PeX+KIRIN (original) and PeX+TYPE both report 21 bugs (a subset of 36 bugs detected with `allyesconfig`). Though PeX+TYPE can capture them all (as type-based analysis is sound yet imprecise), it generates up to 3x more warnings, placing

a high burden on the users side for their manual review. On the other hand, as an unsound solution, PeX+KM produces a limited number of warnings, resulting in the detection of only 6 bugs missing the rest.

6.6.4 Manual Review of Warnings

The manual review process of reported warnings is to determine whether a privileged function identified by PeX (Section 6.5.3) is a *true* privileged function or not. As long as one can confirm that a function is indeed privileged, reported warnings regarding its missing, inconsistent, and redundant permission checks should be *true positives* from PeX's point of view.

Though kernel developers with domain knowledge may be able to discern them with no complication, we (as a third-party) try to understand whether a given function can be used to access critical resources (e.g., device, file system, etc.). As a result, we conservatively reported 36 bug warnings to the community; we suspect that there could be more true warnings missed due to our lack of domain knowledge. We plan to release PeX and the list of potential privileged functions, hoping kernel developers will contribute to identify privileged functions and fix more true permission errors.

Certain static paths reported by PeX may not be feasible dynamically during program execution, resulting in false positives. One may devise a solution solving path constraints as in symbolic execution engines [50] to address this problem, PeX currently does not do so.

Table 6.5: Bugs Reported By PeX. **C**onfirmed or **I**gnored.

Type-#	File	Function	Description	Status
DAC-1	fs/btrfs/send.c	`btrfs_send`	missing DAC check when traversing a snapshot	C
DAC-2	fs/ecryptfs/inode.c	`ecryptfs_removexattr()`,`_setxattr()`	missing `xattr_permission()`	C
DAC-3	fs/ecryptfs/inode.c	`ecryptfs_listxattr()`	missing `xattr_permission()`	C
CAP-4	drivers/char/random.c	`write_pool()`, `credit_entropy_bits()`	missing `CAP_SYS_ADMIN`	C
CAP-5	drivers/scsi/sg.c	`sg_scsi_ioctl()`	missing `CAP_SYS_ADMIN` or `CAP_RAW_IO`	I
CAP-6	drivers/block/pktcdvd.c	`add_store()`, `remove_store()`	missing `CAP_SYS_ADMIN`	I
CAP-7	drivers/char/nvram.c	`nvram_write()`	missing `CAP_SYS_ADMIN`	I
CAP-8	drivers/firmware/efi/efivars.c	`efivar_entry_set()`	missing `CAP_SYS_ADMIN`	C
CAP-9	net/rfkill/core.c	`rfkill_set_block()`, `rfkill_fop_write()`	missing `CAP_NET_ADMIN`	C
CAP-10	block/scsi_ioctl.c	`mmc_rpmb_ioctl()`	missing verify_command or `CAP_SYS_ADMIN`	I
CAP-11	drivers/platform/x86/thinkpad_acpi.c	`acpi_evalf()`	missing `CAP_SYS_ADMIN`	I
CAP-12	drivers/md/dm.c	`dm_blk_ioctl()`	missing `CAP_RAW_IO`	I
CAP-13	block/bsg.c	`bsg_ioctl`	inconsistent/missing `CAP_SYS_ADMIN`	C
CAP-14	kernel/sys.c	`prctl_set_mm_exe_file`	inconsistent capability check	I
CAP-15	kernel/sys.c	`prctl_set_mm_exe_file`	inconsistent capability and namespace check	I
CAP-16	block/scsi_ioctl.c	`blk_verify_command`	redundant check `CAP_SYS_RAWIO`	I
LSM-17	fs/ecryptfs/inode.c	`ecryptfs_removexattr()`, `_setxattr()`	missing `security_inode_removexattr()`	C
LSM-18	mm/mmap.c	`remap_file_pages`	missing `security_mmap_file()`	I
LSM-19	fs/binfmt_elf.c	`load_elf_binary()`	missing `security_kernel_read_file`	I
LSM-20	fs/binfmt_elf.c	`load_elf_library()`	missing `security_kernel_read_file`	I
LSM-21	fs/xfs/xfs_ioctl.c	`xfs_file_ioctl()`	missing `security_inode_readlink()`	C
LSM-22	kernel/workqueue.c	`wq_nice_store()`	missing `security_task_setnice()`	C
LSM-23	fs/ecryptfs/inode.c	`ecryptfs_listxattr()`	missing `security_inode_listxattr`	C
LSM-24	include/linux/sched.h	`comm_write()`	missing `security_task_prctl()`	C
LSM-25	fs/binfmt_misc.c	`load_elf_binary()`	missing `security_bprm_set_creds()`	I
LSM-26	drivers/android/binder.c	`binder_set_nice`	missing `security_task_setnice()`	I
LSM-27	fs/ocfs2/cluster/tcp.c	`o2net_start_listening()`	missing `security_socket_bind`	I
LSM-28	fs/ocfs2/cluster/tcp.c	`o2net_start_listening()`	missing `security_socket_listen`	I
LSM-29	fs/dlm/lowcomms.c	`tcp_create_listen_sock`	missing `security_socket_bind`	I
LSM-30	fs/dlm/lowcomms.c	`tcp_create_listen_sock`	missing `security_socket_listen`	I
LSM-31	fs/dlm/lowcomms.c	`sctp_listen_for_all`	missing `security_socket_listen`	I
LSM-32	net/socket.c	`kernel_bind`	missing `security_socket_bind`	I
LSM-33	net/socket.c	`kernel_listen`	missing `security_socket_listen`	I
LSM-34	net/socket.c	`kernel_connect`	missing `security_socket_connect`	I
LSM-35	fs/ocfs2/cluster/tcp.c	`o2net_start_listening()`	redundant `security_socket_create`	C
LSM-36	fs/ocfs2/cluster/tcp.c	`o2net_open_listening_sock()`	redundant `security_socket_create`	C

6.6.5 Discussion of Security Bug Findings

Missing Check

Fig. 6.2b is one of the confirmed missing LSM checks (LSM-21). We discuss two more confirmed cases.

The CAP-4 missing check in kernel `random` device driver is particularly critical and triggered active discussion in the kernel developer community (including Torvalds). Random number generator serves as the foundation of many cryptography libraries including OpenSSL, and thus the quality of the random number is very critical. This security bug allows attackers to manipulate entropy pool, which can potentially corrupt many applications using cryp-

tography libraries. Specifically, a problematic path starts from `evdev_write` and reaches the privileged function `credit_entropy_bits`, which can control the entropy in the entropy pool, while bypassing the required `CAP_SYS_ADMIN` permission check.

The LSM-21 missing check in `xfs_file_ioctl` led to another interesting discussion among kernel developers [21]. With this interface, a userspace program may perform low-level file system operations, but `security_inode_read_link` LSM hook was missing. An adversary could exploit this interface and gain access to the whole file system that is not allowed by LSM policy. Interestingly, however, the privileged function performed `CAP_SYS_ADMIN` Capability permission check. This created disagreement between kernel developers: one group argues that the LSM hook is necessary, while another thinks that `CAP_SYS_ADMIN` is sufficient. We agree with the former because LSM is designed to limit the damage of a compromised process to the system, even the ones of root user [186]. We believe that LSM permission checks should still be enforced as always for better security even when the current user is root.

Kernel developers decided not to fix 9 of our reports because they believe these bugs are not exploitable. As discussed earlier, PeX in the current form neither validates if a suspicious static path is dynamically reachable, nor generates a concrete exploit to demonstrate the security issue; we believe both are good future works. Nonetheless, we have one complaint to share.

For the LSM-19 and LSM-20 cases, PeX found that the LSM hooks `security_kernel_read_file` and `security_kernel_post_read_file` were used to protect the privileged functions `kernel_read_file` and `kernel_post_read_file` in some program paths. We reported missing LSM hooks in `load_elf_binary` and `load_elf_library` for these privileged functions. However, the kernel developers responded that those hooks are used to monitor loading firmware/kernel modules only (not other files), and thus no patch is required. Here, the implication we found

is three-fold. First, the permission check names are ambiguous and misleading. Second, we were not able to find any documentation of such LSM specification regarding the protection of firmware/kernel modules. Last, PeX actually found a counter-example in `IMA` where the same checks are indeed used for loading other files (neither firmware nor kernel modules). Consequently, we suggest changing the function name and documenting the clear intention to avoid any confusion and to prevent system administrators from creating an LSM policy that does not work.

Inconsistent Check

The CAP-13 inconsistent check has been discussed in Fig. 6.1. One program path in Figs. 6.1a and 6.1c has two `CAP_SYS_RAWIO` checks and one `CAP_SYS_ADMIN` check, while another path in Figs. 6.1b and 6.1c has only one `CAP_SYS_ADMIN` check. PeX detects this bug as an inconsistent check, but from the viewpoint of correction, which requires adding `CAP_SYS_RAWIO`, this may also be viewed as a missing check. There is a separate redundant check error in `CAP_SYS_RAWIO`.

Upon further investigation, we were interested in learning the practices in using multiple capabilities together. `scsi_ioctl` in Fig. 6.1a checks both `CAP_SYS_ADMIN` **and** `CAP_SYS_RAWIO`. However, in a different network subsystem (not shown), we found that `too_many_unix_fds` performs a *weaker* permission check with the `CAP_SYS_ADMIN` **or** `CAP_SYS_RAWIO` condition. We believe this OR-based weaker check is not a good practice because this in effect makes `CAP_SYS_ADMIN` too powerful (like root), diminishing the benefit of fine-grained capability-based access control.

The CAP-14 and CAP-15 inconsistent error reports were acknowledged but ignored by the kernel developers for the following reason. For the same privileged function `prctl_set_mm_exe_file`,

which is used to set an executable file, PeX discovered one case requiring `CAP_SYS_RESOURCE` in `user namespace`, and another case checking `CAP_SYS_ADMIN` in `init namespace`. Kernel developers responded that each case is fine by design for that specific context. PeX does not consider the precise context in which `prctl_set_mm_exe_file` is used (similar to aforementioned `security_kernel_read_file` used for loading kernel modules), leading to an imprecise report, but we believe that both CAP-14 and CAP-15 are worthwhile for further investigation.

Redundant Check

A redundant check occurs in two forms. **First**, for user-reachable functions, it happens when a privileged function is covered by the same permission checks multiple times. We reported three cases. The CAP-16 case was discussed in Figs. 6.1a and 6.1c with two `CAP_SYS_RAWIO` checks, which was ignored by kernel developers. On the other hand, for the LSM-35 and LSM-36 cases found in the `ocfs2` file system, the other kernel developer group confirmed and promised to fix the bugs. **Second**, any permission check in kernel-initialization functions is marked as redundant because the boot thread is executed under root. PeX detected tens of such cases, but we did not report them as they are less critical.

6.7 Summary

In this chapter, we tackle scalability and precision problems in static kernel bug detectors (**PS3**). We presents PeX, a static permission check analysis framework for Linux kernel, which can automatically infer mappings between permission checks and privileged functions as well as detect missing, inconsistent, and redundant permission checks for any privileged functions. PeX relies on KIRIN, our novel call graph analysis based on kernel interfaces, a

common programming pattern in the Linunx kernel, to resolve indirect calls precisely and efficiently (**TS3**).

We evaluated both KIRIN and PeX for the latest stable Linux kernel v4.18.5. The experiments show that KIRIN can resolve 86%-92% of all indirect callsites in the kernel within 7 minutes. In particular, PeX reported 36 permission check bugs of DAC, Capabilities, and LSM, 14 of which have already been confirmed by the kernel developers. PeX is published at [233].

Chapter 7

Conclusion

Addressing software bugs is becoming a more urgent task. Over the years, software bugs are not only causing more and more damages to the economy but affecting people's daily life. Software developers use bug detectors to help locate and fix bugs. However, many bug detectors are not practical, limiting the wide adoption. In this thesis, we study and address common problems of dynamic and static bug detectors, therefore helping developers to improve software quality.

First, we study dynamic bug detectors and address run-time (**PS1**) and space overheads problems (**PS2**). To reduce run-time overhead, we propose to repurpose commodity hardware (**TS1**) to improve dynamic bug detectors. Especially, we study data race bugs, which are becoming popular in multi-threaded shared-memory programs. We design TxRace, which demonstrates that we can leverage commodity transactional memory to accelerate a dynamic data race detector. The experimental results show that TxRace reduces the average run-time overhead from 11.68x to 4.65x, with only a small number of false negatives.

Due to the undeterministic nature of data races, it is hard to find all data races during testing. Thus people are interested in deploying dynamic data race detector in production environment, which can better exercise the program and catch more data races. We present a production-ready sampling-based dynamic data race detector, ProRace, which further reduces the run-time overhead leveraging PMU for lightweight tracing. The evaluation shows that ProRace incurs only 2.6% run-time overhead with 27.5% detection probability with a

sampling period of 10,000.

To address the space overhead problem, we propose to reuse existing metadata maintained by one bug detector to detect other kinds of bugs (**TS2**), reducing space overhead in dynamic bug detectors. We study memory safety bugs, which is more common than data race bugs. We present a memory-efficient solution, BOGO, which adds temporal memory safety upon Intel MPX's spatial memory safety solution. BOGO reuses Intel MPX's bound metadata as well as run-time checks, reducing spacing overhead. The evaluation shows that BOGO incurs 60% run-time overhead and 36% memory overhead providing full memory safety.

Finally, we study static bug detectors and address scalability and precision problems (**PS3**). To improve scalability and precision, we propose applying common programming patterns to static analyses (**TS3**). We focus on static analysis for large program code, the Linux kernel, and improve the scalability and precision of a common analysis, kernel call graph generation, using function pointer usage pattern in the Linux kernel. Furthermore, to demonstrate its effectiveness, we designed a kernel permission check bug detector based on the aforementioned analysis. Our detector can help kernel developers find out missing,inconsistent, and redundant permission checks, that is critical to operating system security. We evaluated it using the latest Linux kernel and it detected 14 previously unknown bugs within a limited time budget.

Bibliography

[1] The apache http server. http://httpd.apache.org.

[2] Cve-2006-1856. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-1856, .

[3] Cve-2011-4080. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-4080, .

[4] CVE-2017-17450 Missing Capability Check in Linux Kernel. https://www.cvedetails.com/cve/CVE-2017-17450/, .

[5] 2011 cwe/sans top 25 most dangerous software errors. http://cwe.mitre.org/top25/.

[6] Child support it failures savaged. https://www.zdnet.com/article/child-support-it-failures-savaged/.

[7] Intel mpx explained – performance evaluation. https://intel-mpx.github.io/performance/.

[8] Nist software assurance reference dataset project. https://samate.nist.gov/SARD.

[9] Android Permission Overview. https://developer.android.com/guide/topics/permissions/overview.

[10] ab - apache http server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

[11] Apparmor. https://gitlab.com/apparmor/apparmor/wikis/home/.

[12] Hack flasback: The mt.gox hack – the most iconic exchange hack. https://www.ledger.com/hack-flasback-the-mt-gox-hack-the-most-iconic-exchange-hack/.

[13] Struct bound narrowing. https://gcc.gnu.org/wiki/Intel MPX support in the GCC compiler#Narrowing.

[14] alse boundaries and arbitrary code execution. https://forums.grsecurity.net/viewtopic.php?f=7&t=2522, .

[15] CAP_SYS_ADMIN: the new root. https://lwn.net/Articles/486306/, .

[16] capabilities - overview of linux capabilities. http://man7.org/linux/man-pages/man7/capabilities.7.html, .

[17] Discretionary access control. https://en.wikipedia.org/wiki/Discretionary_access_control.

[18] Dangsan open source implementation. https://github.com/vusec/dangsan.

[19] K-miner: Data-flow analysis for the linux kernel. https://github.com/ssl-tud/k-miner.

[20] Knight shows how to lose $440 million in 30 minutes. https://www.bloomberg.com/news/articles/2012-08-02/knight-shows-how-to-lose-440-million-in-30-minutes.

[21] Re: Leaking path in xfs's ioctl interface(missing lsm check) by stephen smalley. https://lkml.org/lkml/2018/9/26/668.

[22] Locationmanager. https://developer.android.com/reference/android/location/LocationManager#getLastKnownLocation(java.lang.String).

[23] Mandatory access control. https://en.wikipedia.org/wiki/Mandatory_access_control.

[24] Softbound+cets open source implementation. https://github.com/santoshn/softboundcets-34.

[25] Smatch: pluggable static analysis for c. https://lwn.net/Articles/691882/.

[26] Software fail watch: 5th edition. https://www.tricentis.com/resources/software-fail-watch-5th-edition/.

[27] Sparse. https://www.kernel.org/doc/html/v4.14/dev-tools/sparse.html.

[28] Spec2006 addresssanitizer patch. https://github.com/google/sanitizers/blob/master/address-sanitizer/spec/spec2006-asan.patch.

[29] Scriptable database and system performance benchmark. https://github.com/akopytov/sysbench.

[30] Toyota's killer firmware: Bad design and its consequences. https://www.edn.com/design/automotive/4423428/Toyota-s-killer-firmware--Bad-design-and-its-consequences.

[31] Virtual file system. https://en.wikipedia.org/wiki/Virtual_file_system.

[32] Forgot your windows 98 password? no problem. https://imgur.com/fqjnK.

[33] Whole Program LLVM: a wrapper script to build whole-program llvm bitcode files. https://github.com/travitch/whole-program-llvm.

[34] Yehuda Afek, Amir Levy, and Adam Morrison. Software-improved hardware lock elision. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 212–221, 2014. ISBN 978-1-4503-2944-6.

[35] Periklis Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *USENIX Security Symposium*, pages 177–192, 2010.

[36] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 51–66, 2009.

[37] A. Alomary et al. PEAS-I: A hardware/software co-design system for ASIPs. pages 2–7, 1993.

[38] Gautam Altekar and Ion Stoica. Odr: output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 193–206. ACM, 2009.

[39] Jonathan Anderson, Robert NM Watson, David Chisnall, Khilan Gudka, Ilias Marinos, and Brooks Davis. Tesla: temporally enhanced system logic assertions. In *Proceedings of the Ninth European Conference on Computer Systems*, page 19. ACM, 2014.

[40] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012.

[41] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proc. of the 17th PACT*, October 2008.

[42] C.M. Bishop et al. *Pattern recognition and machine learning*. Springer New York:, 2006.

[43] Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Brandon Lucia. Efficient, software-only data race exceptions. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '15, 2015.

[44] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40. ACM, 2011.

[45] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 81–91, 2007. ISBN 978-1-59593-706-3.

[46] Hans-J. Boehm and Sarita V. Adve. Foundations of the c++ concurrency memory model. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 68–78, 2008.

[47] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. Pacer: Proportional detection of data races. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 255–268, 2010. ISBN 978-1-4503-0019-3.

[48] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 265–275, 2003. ISBN 0-7695-1913-X.

[49] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Pro-*

ceedings of the 2012 International Symposium on Software Testing and Analysis, pages 133–143. ACM, 2012.

[50] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[51] Irina Calciu, Justin Gottschlich, Tatiana Shpeisman, Gilles Pokam, and Maurice Herlihy. Invyswell: A hybrid transactional memory for haswell's restricted transactional memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 187–200, 2014.

[52] Christopher D. Carothers, Kalyan S. Perumalla, and Richard M. Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Trans. Model. Comput. Simul.*, 9(3):224–253, July 1999. ISSN 1049-3301. doi: 10.1145/347823.347828. URL http://doi.acm.org/10.1145/347823.347828.

[53] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 559–572, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0245-6. doi: 10.1145/1866307.1866370. URL http://doi.acm.org/10.1145/1866307.1866370.

[54] Shimin Chen, Babak Falsafi, Phillip B. Gibbons, Michael Kozuch, Todd C. Mowry, Radu Teodorescu, Anastassia Ailamaki, Limor Fix, Gregory R. Ganger, Bin Lin, and Steven W. Schlosser. Log-based architectures for general-purpose monitoring of deployed code. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, pages 63–65, New York,

NY, USA, 2006. ACM. ISBN 1-59593-576-2. doi: 10.1145/1181309.1181319. URL http://doi.acm.org/10.1145/1181309.1181319.

[55] Xi Chen, Asia Slowinska, and Herbert Bos. On the detection of custom memory allocators in c binaries. *Empirical Softw. Engg.*, 21(3):753–777, June 2016. ISSN 1382-3256. doi: 10.1007/s10664-015-9362-z. URL http://dx.doi.org/10.1007/s10664-015-9362-z.

[56] Lee Chew and David Lie. Kivati: Fast detection and prevention of atomicity violations. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 307–320, 2010. ISBN 978-1-60558-577-2.

[57] David Chisnall, Colin Rothwell, Robert NM Watson, Jonathan Woodruff, Munraj Vadera, Simon W Moore, Michael Roe, Brooks Davis, and Peter G Neumann. Beyond the pdp-11: Architectural support for a memory-safe c abstract machine. In *ACM SIGPLAN Notices*, volume 50, pages 117–130. ACM, 2015.

[58] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 258–269, 2002. ISBN 1-58113-463-0.

[59] Jim Chow, Tal Garfinkel, and Peter M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1404014.1404015.

[60] Cliff Click. Azuls experiences with hardware transactional memory. In *In HP Labs - Bay Area Workshop on Transactional Memory*, 2009.

[61] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0036-0. doi: 10.1145/1807128.1807152. URL http://doi.acm.org/10.1145/1807128.1807152.

[62] 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 *Usenix Security*, volume 98, pages 63–78, 1998.

[63] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P. Kemerlis. Retracer: Triaging crashes by reverse execution from partial memory dumps. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 820–831, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3900-1. doi: 10.1145/2884781.2884844. URL http://doi.acm.org/10.1145/2884781.2884844.

[64] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computer Science and Engineering*, 5(1):46–55, 1998.

[65] Luke Dalessandro, François Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael F. Spear. Hybrid norec: A case study in the effectiveness of best effort hardware transactional memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 39–52, 2011. ISBN 978-1-4503-0266-1.

[66] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: Architectural support for spatial safety of the c programming language. In *Proceedings of the 13th International Conference on Architectural Support for Programming*

Languages and Operating Systems, ASPLOS XIII, pages 103–114, 2008. ISBN 978-1-59593-958-6.

[67] Joseph Devietti, Benjamin P. Wood, Karin Strauss, Luis Ceze, Dan Grossman, and Shaz Qadeer. Radish: Always-on sound and complete ra detection in software and hardware. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 201–212, 2012. ISBN 978-1-4503-1642-2.

[68] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for c with very low overhead. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 162–171, 2006. ISBN 1-59593-375-1.

[69] Dinakar Dhurjati and Vikram Adve. Efficiently detecting all dangling pointer uses in production servers. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 269–280. IEEE, 2006.

[70] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. Safecode: Enforcing alias analysis for weakly typed languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 144–157, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4. doi: 10.1145/1133981.1133999. URL http://doi.acm.org/10.1145/1133981.1133999.

[71] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 157–168, 2009. ISBN 978-1-60558-406-5.

[72] Mark Dowson. The ariane 5 software failure. *ACM SIGSOFT Software Engineering Notes*, 22(2):84, 1997.

[73] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Demand-driven computation of interprocedural data flow. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 37–48. ACM, 1995.

[74] Antony Edwards, Trent Jaeger, and Xiaolan Zhang. Runtime verification of authorization hook placement for the linux security modules framework. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 225–234. ACM, 2002.

[75] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J. Boehm. Ifrit: Interference-free regions for dynamic data-race detection. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 467–484, 2012. ISBN 978-1-4503-1561-6.

[76] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: A race and transaction-aware java runtime. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 245–255, 2007. ISBN 978-1-59593-633-2.

[77] 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 *ACM SIGOPS Operating Systems Review*, volume 35, pages 57–72. ACM, 2001.

[78] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *In Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI '10, 2010.

[79] Cormac Flanagan and Stephen N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming*

Language Design and Implementation, PLDI '09, pages 121–133, 2009. ISBN 978-1-60558-392-1.

[80] Vinod Ganapathy, Trent Jaeger, and Somesh Jha. Automatic placement of authorization hooks in the linux security modules framework. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 330–339. ACM, 2005.

[81] Vinod Ganapathy, Trent Jaeger, and Somesh Jha. Towards automated authorization policy enforcement. In *Proceedings of Second Annual Security Enhanced Linux Symposium*. Citeseer, 2006.

[82] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. K-miner: Uncovering memory corruption in linux. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2018.

[83] Bhavishya Goel, Ruben Titos-Gil, Anurag Negi, Sally A. McKee, and Per Stenstrom. Performance and energy analysis of the restricted transactional memory implementation on haswell. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 615–624, Washington, DC, USA, 2014. ISBN 978-1-4799-3800-1.

[84] Google. Addresssanitizeruseafterreturn, 2017. URL https://github.com/google/sanitizers/wiki/AddressSanitizerUseAfterReturn.

[85] Joseph L. Greathouse, Zhiqiang Ma, Matthew I. Frank, Ramesh Peri, and Todd Austin. Demand-driven software race detection using hardware performance counters. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 165–176, 2011. ISBN 978-1-4503-0472-6.

[86] Part Guide. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part*, 2, 2011.

[87] Rajiv Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 2(1-4):135–150, 1993.

[88] Shantanu Gupta, Florin Sultan, Srihari Cadambi, Franjo Ivancic, and Martin Roetteler. Racetm: Detecting data races using transactional memory. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 104–106, 2008. ISBN 978-1-59593-973-9.

[89] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *ACM SIGPLAN Notices*, volume 42, pages 290–299. ACM, 2007.

[90] Ben Hardekopf and Calvin Lin. Exploiting pointer and location equivalence to optimize pointer analysis. pages 265–280, 2007.

[91] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 289–298. IEEE Computer Society, 2011.

[92] Ruud Haring, Martin Ohmacht, Thomas Fox, Michael Gschwind, David Satterfield, Krishnan Sugavanam, Paul Coteus, Philip Heidelberger, Matthias Blumrich, Robert Wisniewski, alan gara, George Chiu, Peter Boyle, Norman Chist, and Changhoan Kim. The ibm blue gene/q compute chip. *IEEE Micro*, 32(2):48–60, March 2012. ISSN 0272-1732.

[93] William Hasenplaugh, Andrew Nguyen, and Nir Shavit. Quantifying the capacity

limitations of hardware transactional memory. In *WTTM '15: 7th Workshop on the Theory of Transactional Memory*, July 2015.

[94] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, 1993. ISBN 0-8186-3810-9.

[95] Jeff Huang, Charles Zhang, and Julian Dolby. Clap: Recording local executions to reproduce concurrency failures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 141–152, 2013. ISBN 978-1-4503-2014-6.

[96] Intel. Intel architecture instruction set extensions programming reference. chapter 8: Intel transactional synchronization extensions, 2012. https://software.intel.com/sites/default/files/m/9/2/3/41604.

[97] Intel. Intel 64 and ia-32 architectures optimization reference manual. chapter 12: Intel tsx recommendations, 2014. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[98] Intel. Intel inspector xe, 2015. http://software.intel.com/en-us/intel-inspector-xe.

[99] *Intel®Microarchitecture Codename Nehalem Performance Monitoring Unit Programming Guide*. Intel Corporation, 2010.

[100] *6th Generation Intel®Processor Datasheet for S-Platforms*. Intel Corporation, 2015.

[101] *Intel®64 and IA-32 Architectures Software Developers' Manual*. Intel Corporation, Santa Clara, CA, 2016.

[102] International Organization for Standardization. ISO/IEC 14882:2011: Information technology – Programming languages – C++, 2011.

[103] International Organization for Standardization. ISO/IEC 9899:2011: Information technology – Programming languages – C, 2011.

[104] Christian Jacobi, Timothy Slegel, and Dan Greiner. Transactional memory architecture and implementation for ibm system z. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 25–36, 2012. ISBN 978-0-7695-4924-8.

[105] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[106] Guoliang Jin, Aditya V. Thakur, Ben Liblit, and Shan Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *OOPSLA*, 2010.

[107] jndok. Analysis and exploitation of pegasus kernel vulnerabilities. http://jndok.github.io/2016/10/04/pegasus-writeup/.

[108] C. Jung, D. Lim, J. Lee, and S. Han. Adaptive execution techniques for SMT multiprocessor architectures. In *ppopp05*, pages 236–246, 2005.

[109] Changhee Jung, Silvius Rus, Brian P. Railing, Nathan Clark, and Santosh Pande. Brainy: effective selection of data structures. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 86–97, New York, NY, USA, 2011. ACM.

[110] Changhee Jung, Sangho Lee, Easwaran Raman, and Santosh Pande. Automated memory leak detection for production use. In *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[111] Baris Kasikci, Cristian Zamfir, and George Candea. Racemob: Crowdsourced data race detection. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 406–422, 2013. ISBN 978-1-4503-2388-8.

[112] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 344–360, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3834-9. doi: 10.1145/2815400.2815412. URL http://doi.acm.org/10.1145/2815400.2815412.

[113] Kirk Kelsey, Tongxin Bai, Chen Ding, and Chengliang Zhang. Fast track: A software system for speculative program optimization. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 157–168, 2009. ISBN 978-0-7695-3576-0.

[114] Bogdan Korel and Juergen Rilling. Program slicing in understanding of large programs. In *Program Comprehension, 1998. IWPC'98. Proceedings., 6th International Workshop on*, pages 145–152. IEEE, 1998.

[115] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, 2004. ISBN 0-7695-2102-9.

[116] Michael A. Laurenzano, Mustafa M. Tikir, Laura Carrington, and Allan Snavely. Pebil: Efficient static binary instrumentation for linux. In *International Symposium on the Performance Analysis of Systems and Software*, 2010.

[117] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long

Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS*, 2015.

[118] Dongyoon Lee, Mahmoud Said, Satish Narayanasamy, Zijiang Yang, and Cristiano Pereira. Offline symbolic analysis for multi-processor execution replay. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 564–575, 2009. ISBN 978-1-60558-798-1.

[119] Dongyoon Lee, Mahmoud Said, Satish Narayanasamy, and Zijiang Yang. Offline symbolic analysis to infer total store order. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 357–358, 2011. ISBN 978-1-4244-9432-3.

[120] Dongyoon Lee, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Chimera: Hybrid program analysis for determinism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 463–474, 2012. ISBN 978-1-4503-1205-9.

[121] Jaejin Lee, Jung-Ho Park, Honggyu Kim, Changhee Jung, Daeseob Lim, and SangYong Han. Adaptive execution techniques of parallel programs for multiprocessors. *J. Parallel Distrib. Comput.*, 70(5):467–480, May 2010. ISSN 0743-7315.

[122] Sangho Lee, Changhee Jung, and Santosh Pande. Detecting memory leaks through introspective dynamic behavior modelling using machine learning. In *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[123] Yosef Lev, Mark Moir, and Dan Nussbaum. PhTM: Phased transactional memory. In *TRANSACT '07: 2nd Workshop on Transactional Computing*, aug 2007.

[124] Nancy G. Leveson and Clark S. Turner. An investigation of the therac-25 accidents. *IEEE Computer*, 26(7):18–41, July 1993.

[125] Qingrui Liu, Changhee Jung, Dongyoon Lee, and Devesh Tiwari. Compiler-directed lightweight checkpointing for fine-grained guaranteed soft error recovery. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2016.

[126] Yutao Liu, Yubin Xia, Haibing Guan, Binyu Zang, and Haibo Chen. Concurrent and consistent virtual machine introspection with hardware transactional memory. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture*, HPCA '14, 2014.

[127] Yutao Liu, Peitao Shi, Xinran Wang, Haibo Chen, Binyu Zang, and Haibing Guan. Transparent and efficient cfi enforcement with intel processor trace. In *Proceedings of the 2017 IEEE 23rd International Symposium on High Performance Computer Architecture*, HPCA '17, 2017.

[128] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. Unisan: Proactive kernel memory initialization to eliminate data leakages. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 920–932. ACM, 2016.

[129] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*, volume 5, 2005.

[130] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of*

the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIII, pages 329–339, 2008. ISBN 978-1-59593-958-6.

[131] Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans-J. Boehm. Conflict exceptions: Simplifying concurrent language semantics with precise hardware exceptions for data-races. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 210–221, 2010. ISBN 978-1-4503-0053-7.

[132] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, 2005. ISBN 1-59593-056-6.

[133] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. Dr. checker: A soundy analysis for linux kernel drivers. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1007–1024. USENIX Association, 2017.

[134] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 378–391, 2005. ISBN 1-58113-830-X.

[135] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. Literace: Effective sampling for lightweight data-race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 134–143, 2009. ISBN 978-1-60558-392-1.

[136] Milo Martin, Colin Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Comput. Archit. Lett.*, 5(2):17–17, July 2006. ISSN 1556-6056.

[137] Hassan Salehe Matar, Ismail Kuru, Serdar Tasiran, and Roman Dementiev. Accelerating precise race detection using commercially available hardware transactional memory support. In *5th Workshop on Determinism and Correctness in Parallel Programming*, WoDet '14, 2014.

[138] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 361–377. ACM, 2015.

[139] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the 2006 IEEE 12th International Symposium on High Performance Computer Architecture*, pages 254–265, February 2006.

[140] S.S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997.

[141] Divya Muthukumaran, Trent Jaeger, and Vinod Ganapathy. Leveraging choice to automate authorization hook placement. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 145–156. ACM, 2012.

[142] Divya Muthukumaran, Nirupama Talele, Trent Jaeger, and Gang Tan. Producing hook placements to enforce expected access control policies. In *International Symposium on Engineering Secure Software and Systems*, pages 178–195. Springer, 2015.

[143] Abdullah Muzahid, Dario Suárez, Shanxiang Qi, and Josep Torrellas. Sigrace: Signature-based data race detection. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 337–348, 2009.

[144] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 245–258, 2009. ISBN 978-1-60558-392-1.

[145] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Cets: compiler enforced temporal safety for c. In *ACM Sigplan Notices*, volume 45, pages 31–40. ACM, 2010.

[146] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 189–200, 2012. ISBN 978-1-4503-1642-2.

[147] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdoglite: Hardware-accelerated compiler-based pointer checking. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 175:175–175:184, 2014. ISBN 978-1-4503-2670-4.

[148] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Everything you want to know about pointer-based checking. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[149] Santosh Ganapati Nagarakatte. *Practical low-overhead enforcement of memory safety for c programs*. PhD thesis, University of Pennsylvania, 2012.

[150] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):477–526, 2005.

[151] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, 2007. ISBN 978-1-59593-633-2.

[152] Robert H. B. Netzer and Barton P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, March 1992. ISSN 1057-4514.

[153] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Efficiently and precisely locating memory leaks and bloat. In *Proc. of the 30th PLDI*, 2009.

[154] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX explained: An empirical study of intel MPX and software-based bounds checking approaches. *CoRR*, abs/1702.00719, 2017. URL http://arxiv.org/abs/1702.00719.

[155] Hilarie Orman. The morris worm: A fifteen-year perspective. *IEEE Security & Privacy*, 1(5):35–43, 2003.

[156] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. In *Acm sigops operating systems review*, volume 42, pages 247–260. ACM, 2008.

[157] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H Lee, and Shan Lu. Pres: probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 177–192. ACM, 2009.

[158] PCWorld. Nasdaq's facebook glitch came from race conditions, May 2012. http://www.pcworld.com/article/255911/ nasdaqs_facebook_glitch_came_from_race_conditions.html.

[159] Fernando Magno Quintao Pereira and Daniel Berlin. Wave propagation and deep propagation for pointer analysis. In *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*, pages 126–135. IEEE, 2009.

[160] Eli Pozniansky and Assaf Schuster. Multirace: Efficient on-the-fly data race detection in multithreaded c++ programs: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(3):327–340, March 2007. ISSN 1532-0626.

[161] Marco Prandini and Marco Ramilli. Return-oriented programming. *IEEE Security & Privacy*, 10(6):84–87, 2012.

[162] Dick Price. Pentium fdiv flaw-lessons learned. *IEEE Micro*, 15(2):86–88, 1995.

[163] Milos Prvulovic. Cord: Cost-effective (and nearly overhead-free) order-recording and data race detection. In *Proceedings of the 2006 IEEE 12th International Symposium on High Performance Computer Architecture*, HPCA '06, 2006.

[164] Milos Prvulovic and Josep Torrellas. Reenact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, pages 110–121, 2003. ISBN 0-7695-1945-8.

[165] Feng Qin, Shan Lu, and Yuanyuan Zhou. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 291–302. IEEE, 2005.

[166] Lili Qiu, Yin Zhang, Feng Wang, Mi Kyung, and Han Ratul Mahajan. Trusted computer system evaluation criteria. In *National Computer Security Center*. Citeseer, 1985.

[167] Sanjay Rawat, Laurent Mounier, and Marie-Laure Potet. Listt: An investigation into unsound-incomplete yet practical result yielding static taintflow analysis. In *Availability, Reliability and Security (ARES), 2014 Ninth International Conference on*, pages 498–505. IEEE, 2014.

[168] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, 30(4).

[169] Carl G. Ritson and Frederick R.M. Barnes. An evaluation of intel's restricted transactional memory for cpas, 2013.

[170] Simon Rogerson. The chinook helicopter disaster. *IMIS Journal*, 12(2), 2002.

[171] Michiel Ronsse and Koen De Bosschere. Recplay: A fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, May 1999. ISSN 0734-2071.

[172] Olatunji Ruwase and Monica S Lam. A practical dynamic buffer overflow detector. In *NDSS*, volume 2004, pages 159–169, 2004.

[173] Paul Sack, Brian E. Bliss, Zhiqiang Ma, Paul Petersen, and Josep Torrellas. Accurate and efficient filtering for the intel thread checker race detector. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, pages 34–41, 2006. ISBN 1-59593-576-2.

[174] Ravi S Sandhu and Pierangela Samarati. Access control: principle and practice. *IEEE communications magazine*, 32(9):40–48, 1994.

[175] Pawel Sarbinowski, Vasileios P Kemerlis, Cristiano Giuffrida, and Elias Athanasopoulos. Vtpin: practical vtable hijacking protection for binaries. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 448–459. ACM, 2016.

[176] Madalina-Ioana Sas. Snowwall: A visual firewall for the surveillance society. 2017.

[177] 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, November 1997. ISSN 0734-2071.

[178] Mark Seaborn and Thomas Dullien. Exploiting the dram rowhammer bug to gain kernel privileges. *Black Hat*, pages 7–9, 2015.

[179] SecurityFocus. Software bug contributed to blackout, Feb. 2004. http://www.securityfocus.com/news/8016.

[180] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, 2009. ISBN 978-1-60558-793-6.

[181] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[182] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 28–28, Berkeley,

CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2342821.2342849.

[183] Yuru Shao, Qi Alfred Chen, Zhuoqing Morley Mao, Jason Ott, and Zhiyun Qian. Kratos: Discovering inconsistent security policy enforcement in the android framework. In *NDSS*, 2016.

[184] Tianwei Sheng, Neil Vachharajani, Stephane Eranian, Robert Hundt, Wenguang Chen, and Weimin Zheng. Racez: A lightweight and non-invasive race detection tool for production applications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 401–410, 2011. ISBN 978-1-4503-0445-0.

[185] Robert Skeel. Roundoff error and the patriot missile. *SIAM News*, 25(4):11, 1992.

[186] Stephen Smalley and Robert Craig. Security enhanced (se) android: Bringing flexible mac to android. In *NDSS*, volume 310, pages 20–38, 2013.

[187] Stephen Smalley, Chris Vance, and Wayne Salamon. Implementing selinux as a linux security module. *NAI Labs Report*, 1(43):139, 2001.

[188] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. Rolecast: finding missing security checks when you do not know what checks are. In *ACM SIGPLAN Notices*, volume 46, pages 1069–1084. ACM, 2011.

[189] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 265–266. ACM, 2016.

[190] Yulei Sui, Ding Ye, Yu Su, and Jingling Xue. Eliminating redundant bounds checks in dynamic buffer overflow detection using weakest preconditions. *IEEE Transactions on Reliability*, 65(4):1682–1699, 2016.

[191] Martin Susskraut, Thomas Knauth, Stefan Weigert, Ute Schiffel, Martin Meinhold, and Christof Fetzer. Prospect: A compiler framework for speculative parallelization. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 131–140, 2010.

[192] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 48–62. IEEE, 2013.

[193] Lin Tan, Xiaolan Zhang, Xiao Ma, Weiwei Xiong, and Yuanyuan Zhou. Autoises: Automatically inferring security specification and detecting violations. In *USENIX Security Symposium*, pages 379–394, 2008.

[194] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. 2002.

[195] The Clang Team. Clang 3.8 threadsanitizer, 2015. http://clang.llvm.org/docs/ThreadSanitizer.html.

[196] Frank Tip. *A survey of program slicing techniques*. Centrum voor Wiskunde en Informatica, 1994.

[197] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. On the expressiveness of return-into-libc attacks. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, RAID'11, pages 121–141, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-23643-3. doi: 10.1007/978-3-642-23644-0_7. URL http://dx.doi.org/10.1007/978-3-642-23644-0_7.

[198] National Computer Security Center (US). *A guide to understanding discretionary access control in trusted systems*, volume 3. National Computer Security Center, 1987.

[199] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable use-after-free detection. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 405–419. ACM, 2017.

[200] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1675–1689, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4139-4. doi: 10.1145/2976749.2978406. URL http://doi.acm.org/10.1145/2976749.2978406.

[201] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Doubleplay: Parallelizing sequential logging and replay. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 15–26, 2011. ISBN 978-1-4503-0266-1.

[202] Guru Venkataramani, Brandyn Roemer, Yan Solihin, and Milos Prvulovic. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 273–284. IEEE, 2007.

[203] Hayawardh Vijayakumar, Xinyang Ge, Mathias Payer, and Trent Jaeger. Jigsaw: Protecting resource access by inferring programmer expectations. In *USENIX Security Symposium*, pages 973–988, 2014.

[204] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Relay: Static race detection on millions of lines of code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The*

Foundations of Software Engineering, ESEC-FSE '07, pages 205–214, 2007. ISBN 978-1-59593-811-4.

[205] Jaroslav Ševčík and David Aspinall. On validity of program transformations in the java memory model. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 27–51, 2008. ISBN 978-3-540-70591-8.

[206] Pengfei Wang, Jens Krinke, Kai Lu, Gen Li, and Steve Dodier-Lazaro. How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the linux kernel. In *USENIX Security Symposium*, 2017.

[207] Wenwen Wang, Kangjie Lu, and Pen-Chung Yew. Check it again: Detecting lacking-recheck bugs in os kernels. In *Proceedings of ACM conference on Computer and communications security*. ACM, 2018.

[208] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. Improving integer security for systems with kint. In *OSDI*, volume 12, pages 163–177, 2012.

[209] Xiaoxiao Wang, Mohammad Tehranipoor, and Jim Plusquellic. Detecting malicious inclusions in secure hardware: Challenges and solutions. In *Hardware-Oriented Security and Trust, 2008. HOST 2008. IEEE International Workshop on*, pages 15–19. IEEE, 2008.

[210] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37. IEEE, 2015.

[211] Benjamin Wester, David Devecsery, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Parallelizing data race detection. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 27–38, 2013. ISBN 978-1-4503-1870-9.

[212] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. Ripe: Runtime intrusion prevention evaluator. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 41–50, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0672-0. doi: 10.1145/2076732.2076739. URL http://doi.acm.org/10.1145/2076732.2076739.

[213] R.R. Wilcox. *Introduction to Robust Estimation and Hypothesis Testing*. Elsevier Science & Technology, 2012. ISBN 9780123869838.

[214] Chris Wright, Crispin Cowan, James Morris, Stephen Smalley, and Greg Kroah-Hartman. Linux security module framework. In *Ottawa Linux Symposium*, volume 8032, pages 6–16, 2002.

[215] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. Cfimon: Detecting violation of control flow integrity using performance counters. In *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-1-4673-1624-8. URL http://dl.acm.org/citation.cfm?id=2354410.2355130.

[216] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. Precise and scalable detection of double-fetch bugs in os kernels. 2018.

[217] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of*

the 2013 ACM SIGSAC conference on Computer & communications security, pages 499–510. ACM, 2013.

[218] Junfeng Yang, Ted Kremenek, Yichen Xie, and Dawson Engler. Meca: an extensible, expressive system and language for statically checking security properties. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 321–334. ACM, 2003.

[219] Junfeng Yang, Ang Cui, Sal Stolfo, and Simha Sethumadhavan. Concurrency attacks. In *The 4th USENIX Workshop on Hot Topics in Parallelism*, Berkeley, CA, 2012. USENIX. URL https://www.usenix.org/conference/hotpar12/concurrency-attacks.

[220] Kaiyuan Yang, Matthew Hicks, Qing Dong, Todd Austin, and Dennis Sylvester. A2: Analog malicious hardware. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 18–37. IEEE, 2016.

[221] Ding Ye, Yu Su, Yulei Sui, and Jingling Xue. Wpbound: Enforcing spatial memory safety efficiently at runtime with weakest preconditions. In *Proceedings of the 2014 IEEE 25th International Symposium on Software Reliability Engineering*, ISSRE '14, pages 88–99, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-6033-0. doi: 10.1109/ISSRE.2014.20. URL http://dx.doi.org/10.1109/ISSRE.2014.20.

[222] Ding Ye, Yu Su, Yulei Sui, and Jingling Xue. Wpbound: Enforcing spatial memory safety efficiently at runtime with weakest preconditions. In *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*, ISSRE '14, pages 88–99, Washington, DC, USA, 2014. IEEE, IEEE Computer Society. ISBN

978-1-4799-6033-0. doi: 10.1109/ISSRE.2014.20. URL http://dx.doi.org/10.1109/ISSRE.2014.20.

[223] Suan Hsi Yong and Susan Horwitz. Protecting c programs from attacks via invalid pointer dereferences. In *ACM SIGSOFT Software Engineering Notes*, volume 28, pages 307–316. ACM, 2003.

[224] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of intel® transactional synchronization extensions for high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 19:1–19:11, 2013. ISBN 978-1-4503-2378-9.

[225] Yves Younan. Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *NDSS*, 2015.

[226] Jie Yu and Satish Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 325–336, 2009. ISBN 978-1-60558-526-0.

[227] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 221–234, 2005. ISBN 1-59593-079-5.

[228] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. Apisan: Sanitizing api usages through semantic cross-checking. In *USENIX Security Symposium*, pages 363–378, 2016.

[229] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R. Sekar. A platform for secure static binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS*

International Conference on Virtual Execution Environments, VEE '14, pages 129–140, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2764-0. doi: 10.1145/2576195.2576208. URL http://doi.acm.org/10.1145/2576195.2576208.

[230] Tong Zhang, Dongyoon Lee, and Changhee Jung. Txrace: Efficient data race detection using commodity hardware transactional memory. *ACM SIGARCH Computer Architecture News*, 44(2):159–173, 2016.

[231] Tong Zhang, Changhee Jung, and Dongyoon Lee. Prorace: Practical data race detection for production use. *ACM SIGOPS Operating Systems Review*, 51(2):149–162, 2017.

[232] Tong Zhang, Dongyoon Lee, and Changhee Jung. Bogo: Buy spatial memory safety, get temporal memory safety (almost) free. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 631–644. ACM, 2019.

[233] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M Azab, and Ruowen Wang. Pex: A permission check analysis framework for linux kernel. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1205–1220, 2019.

[234] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using cqual for static analysis of authorization hook placement. In *USENIX Security Symposium*, pages 33–48, 2002.

[235] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. Hard: Hardware-assisted lockset-based race detection. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 121–132, 2007. ISBN 1-4244-0804-0.