Parallel Memory Permissions and Their Applications

by

Ali Razeen

Department of Computer Science
Duke University

Date: ________________________
Approved:

____
Landon P. Cox, Supervisor

____
Alvin R. Lebeck

____
Jeffrey S. Chase

____
Daniel J. Sorin

Dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2018

<u>Abstract</u>

Parallel Memory Permissions and Their Applications

by

Ali Razeen

Department of Computer Science
Duke University

Date: ____________________
Approved:

____
Landon P. Cox, Supervisor

____
Alvin R. Lebeck

____
Jeffrey S. Chase

____
Daniel J. Sorin

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University
2018

Copyright © 2018 by Ali Razeen
All rights reserved except the rights granted by the
Creative Commons Attribution-Noncommercial Licence

Abstract

A process can voluntarily set memory protections to different portions of its address space. As threads in a process share the same address space, they are equally bound to its protections. In this dissertation, we explore the concept of *parallel memory permissions*, a powerful technique that allows multiple threads to execute in parallel while having different permissions to the same address space, and we show how it may be implemented on commodity hardware without requiring special hardware primitives. Parallel memory permissions makes it practical and easy to apply various tools and protection schemes in multi-threaded applications; practical because it does not hinder threads from executing in parallel and easy because by virtue of not segmenting a process's address space, applications require only slight modifications (if any) to benefit from it.

We demonstrate this first with SandTrap, a Dynamic Information-Flow Tracking (DIFT) tool for machine code on Android. SandTrap complements TaintDroid and addresses its key limitation: the inability to track information flows when an app calls third-party native functions. The key to SandTrap is on-demand DIFT, where DIFT is performed only when threads access data that needs to be tracked. Otherwise, they run unmodified without any DIFT overhead. As Android apps are inherently multi-threaded, on-demand DIFT in SandTrap is possible because of parallel memory permissions. It allows SandTrap to set different sets of memory protections on a thread depending on whether it is running third-party native code, and it does so

without hindering the parallelism of the app. The current prototype of SandTrap runs on a real smartphone device with unmodified apps downloaded from the Google Play Store, such as Instagram.

Next, we present DoubleVision, a system that addresses the problem of stray memory accesses in which a thread might accidentally read or write to memory-mapped application resources. These stray accesses often occur because of bugs in the application code and they can violate the integrity or confidentiality of resources in memory. We show how DoubleVision uses parallel memory permissions to restrict a thread from inadvertently accessing such resources, without requiring significant refactoring of the application.

To Ana, Landon, and Ben

Contents

Abstract **iv**

List of Tables **x**

List of Figures **xi**

List of Abbreviations **xiii**

1 Introduction **1**

2 Parallel Memory Permissions **3**

2.1 Multiple Page Tables 7

2.1.1 Synchronizing virtual-to-physical mappings 8

2.1.2 Managing the TLB 9

2.1.3 Minimizing the page table size 10

2.2 ARMv7 memory domains 10

2.2.1 Managing copy-on-write 12

2.3 Tradeoffs 13

3 SandTrap: Tracking Information Flows On Demand with Parallel Permissions **14**

3.1 Introduction 14

3.2 Background 16

3.2.1 Android: threads, native code, and shared buffers 16

3.2.2 DIFT: performance and precision 18

3.3 System Overview 24
3.3.1 Design principles 24
3.3.2 Trust and threat model 30
3.4 SandTrap 31
3.4.1 Label storage 31
3.4.2 Parallel memory permissions 33
3.4.3 Thread transitions 34
3.5 Implementation 36
3.5.1 The initial prototype 36
3.5.2 Using MAMBO 40
3.6 Evaluation 42
3.6.1 Experimental methodology 43
3.6.2 On-demand vs continuous DIFT 44
3.6.3 False traps 48
3.6.4 Energy consumption 50
3.6.5 Memory domains vs two page tables 52
3.7 Related work 53
3.8 Conclusion 54
4 DoubleVision: Protecting Application Resources with Parallel Permissions 55
4.1 Introduction 55
4.2 Background 58
4.2.1 Memory-Mapped I/O 58
4.2.2 Stray memory accesses 61
4.2.3 Page protections 63
4.3 Design 63

4.3.1 Focus on buggy stray accesses 63
4.3.2 Amenable to existing applications 64
4.3.3 Compatibility with existing hardware 65
4.3.4 Must not limit parallelism 66
4.4 DoubleVision 67
4.4.1 Parallel memory permissions 67
4.4.2 Types of protections 67
4.4.3 API 68
4.4.4 Limitations 70
4.5 Evaluation 71
4.5.1 Modifying libraries 72
4.5.2 DoubleVision and `mmio` 73
4.5.3 DoubleVision overhead 74
4.5.4 Summary 80
4.6 Related Work 80
4.7 Conclusion 83
5 Conclusion **85**
5.1 Future work 86
Bibliography **88**
Biography **94**

List of Tables

2.1 General API of parallel memory permissions. 7

3.1 Android applications used to test overall performance of SandTrap while tracking camera, location, and microphone data. 43

4.1 DoubleVision API 68

4.2 Summary of our modifications to libraries to protect resources with DoubleVision. 72

4.3 Description of IOZone & LMDB Benchmarks. 73

List of Figures

2.1 An illustration of how tbl_A and tbl_B share second-level page tables where possible, so as to reduce the total size occupied by the page tables. 11

3.1 The frame rate recorded from a dual-core Android smartphone when loading a PDF file in eBooka, an ebook reader app, with parallel and permissions-driven execution. 23

3.2 An illustration of how SandTrap uses parallel memory permissions with ARM memory domains. 28

3.3 Overall slowdown imposed by SandTrap on native code execution in a variety of Android applications (tracked data sources: camera, location, and microphone). 45

3.4 Slowdown imposed by SandTrap on native code execution while tracking different data sources through Instagram. 46

3.5 Individual SandTrap component overhead on native code execution in Instagram. 47

3.6 A measure of how quickly a native routine in Instagram reads tainted data after emulation begins. 49

3.7 Total MAMBO blocks executed in Instagram. 50

3.8 Energy consumed by the device while running the Instagram experiments under different configurations. 51

3.9 Memory consumed by SandTrap while running Instagram experiments under different configurations (tracking camera data). 52

4.1 Speedup of using `mmio` over `rwio` when reading from a file in IOZone benchmarks. 59

4.2 Speedup of using `mmio` over `rwio` when writing to a file. 60

4.3 Speedup of using mapping databases with read-write permissions instead of just read in LMDB. 62

4.4 Speedup of using `mmio` (w/ DoubleVision) over stock `rwio`. 74

4.5 Slowdown of using DoubleVision `mmio` over stock `mmio` when writing to a file. 75

4.6 Slowdown of using DoubleVision `mmio` over stock `mmio` when reading from a file. 76

4.7 Speedup of using LMDB with DoubleVision enabled and the database mapped with write permissions over stock LMDB (database mapped with read-only permissions). 77

4.8 Slowdown of DoubleVision LMDB over stock LMDB when the database is mapped with write permissions in both cases. 77

4.9 Performance of DoubleVision SQLite over stock SQLite without memory-mapping the database. 78

4.10 Slowdown of DoubleVision SQLite over stock SQLite, with memory-mapped databases in both cases. 79

4.11 Performance of nginx with DoubleVision OpenSSL and stock OpenSSL. 80

List of Abbreviations

Abbreviations

ADB	Android Debug Bridge.
ANR	App-not-responding.
AOT	Ahead-of-time.
API	Application Programming Interface.
ART	Android Runtime.
ASID	Address Space Identifier.
AS	Address Space.
App	Application.
CPU	Central Processing Unit.
DACR	Domain Access Control Register.
DIFT	Dynamic Information Flow Tracking.
DOM	Document Object Model.
ELF	Executable and Linkable Format.
I/O	Input/Output.
ISA	Instruction Set Architecture.
IoT	Internet of Things.
JIT	Just-in-time.
JNI	Java Native Interface.
SIMD	Single Instruction, Multiple Data.

TLB	Translation Lookaside Buffer.
UID	User Identifier.
UI	User Interface.
VM	Virtual Machine.

1

Introduction

The virtual memory abstraction [29] provided by modern operating systems (OSes) and CPU platforms allows a process to set memory protections to different portions of its address space (AS). These memory protections are set on a per virtual page basis, and are applied process-wide. In a multi-threaded process, since the different threads share the same AS, they are equally bound to its protections.

In this dissertation, we explore *Parallel Memory Permissions*, a technique that allows threads to execute in parallel while having different permissions to a shared AS. Similar to virtual memory, it enables systems to set page protections in a process's AS. However, it also allows them to decide, during runtime, whether a thread should abide by those protections or ignore them. If parallel memory permissions are not used, the application's default behavior is no different from the existing memory protections scheme OSes already provide.

Parallel memory permissions does not rely on hardware-specific features nor do it alter well-understood OS primitives such as processes, threads, and address spaces. This enables systems to use parallel memory permissions on existing commodity hardware, and it does not require significant code modifications in applications for

them to use it (if at all).

In the rest of this dissertation, we first discuss in detail parallel memory permissions, how it may be implemented, and how it relates to existing work (Chapter 2). Next, we discuss two different systems that use parallel memory permissions as a key building block (Chapter 3 and Chapter 4). Finally, we discuss other potential uses of parallel memory permissions (Chapter 5).

2

Parallel Memory Permissions

Suppose that an application process P, with threads T_m and T_n, has a memory page mapped to virtual address V_x in its AS. Suppose further that a system needs to allow T_m to access V_x but not T_n. Since T_m and T_n share an AS, page protections apply to them equally. The system may either allow or deny both of their accesses to V_x, but it may not give them different permissions to V_x at the same time. The following are some ways of meeting this system's requirements.

Use different ASes

Instead of having a single process with two threads, we can refactor the application into two separate processes: P_m and P_n. Next, we can map the page V_x into P_m's AS but not P_n's. The first issue with this approach is that it requires non-trivial modifications to the application's code. Observe that because threads in a process share an AS, T_m and T_n share and have the same permissions to all pages. The system only requires permissions for V_x to be different. If we split the threads into two separate processes to meet this requirement, we need to explicitly map resources the threads share in both processes' ASes. Such resources include the heap area and

code regions. This is an onerous undertaking, particularly in existing applications with large codebases. Moreover, this may not be possible when we do not have access to the application's source code (see Chapter 3).

The second issue is that the system might require threads to have different permissions to V_x based on their execution contexts. For example, it may require them to have access to V_x while they are executing a well-known routine but not in other cases. If we use separate processes, we need to swap a process's AS based on the code it is running, which is highly non-trivial using existing OS primitives.

This second issue may be addressed with abstractions such as light-weight contexts (lwCs) [46]. Just like processes, each lwC has its own virtual memory mappings and file descriptors. However, unlike processes, they are not schedulable entities; instead a thread chooses an lwC to execute under and may later switch to a different lwC. Unfortunately, lwCs are still subject to the first limitation; by default, each lwC is separate and it is up to the developer to explicitly share resources among different lwCs. Therefore, we still require potentially non-trivial modifications to applications, which, again, may not possible without access to their source code.

Hsu et al. proposed *secure memory views* (SMVs), an approach related to having different ASes [41]. They observe that in a multi-threaded application, a thread compromised by a malicious adversary can read arbitrary regions of the shared AS and access sensitive data components that should only be accessible by other threads. One way of protecting against this threat model is to split the multi-threaded application into multiple processes to enforce process-level isolation between the components that should not be shared. However, like us, they consider this impractical.

Instead, they propose SMVs, which provides thread-level memory isolation. First, the developer will place each contiguous region of memory into a protection domain. Next, she will define SMVs where each SMV has a separate set of permissions to the different protection domains. Finally, she will refactor applications to represent

threads using the `SMVthread` abstraction instead of `pthreads`, and she will also assign each `SMVthread` an SMV. During runtime, all `SMVthreads` execute in parallel, access the shared AS via their respective SMVs, and are prevented from switching to other SMVs. If data needs to be shared between multiple `SMVthreads`, it needs to be done so explicitly. Since the shared AS is now isolated between different SMVs, a compromised `SMVthread` is restricted to only the data accessible from its SMV.

Observe that because an `SMVthread` cannot switch between different SMVs during execution, a thread's permissions to the AS may not be altered depending on its execution context. To repeat our earlier example, there may be a situation where a thread should not have access to V_x unless it is executing some trusted routine. Suppose that this thread starts out in an SMV that does not have access to V_x. Since threads may not switch between different SMVs, it may not access V_x, even when it is executing its trusted routine. One way of addressing this is to refactor the application so that threads do not execute both untrusted and trusted routines, and to offload the task of reading from V_x to a separate thread. This requires access to the applications' source code and may be impractical, depending on the complexity of the application.

Control the thread's execution

We could control how T_m and T_n execute: when T_m runs, allow accesses to V_x and prevent T_n from running. Conversely, when T_n runs, deny accesses to V_x and prevent T_m from running. This may be implemented within the application by modifying it to carefully use synchronization primitives, or by changing the scheduler in the kernel. Its main drawback is that it hinders parallelism, thereby impacting performance. We consider this unacceptable given that developers often write multi-threaded applications precisely for good performance.

Require hardware-specific features

We may use hardware-specific features that are designed expressly for this purpose. For example, Intel's Memory Protection Keys (MPKs) allow threads to have different page protections for the same virtual page in a shared AS [5]. The downside of this approach is that such features are not standardized and are not uniformly available, even within the same ISA. For example, MPK is available only in certain Intel x86 CPUs and it is not clear whether AMD's x86 CPUs will have similar features.

Use alternate memory protection schemes

Existing work in the literature propose a number of different memory protection schemes, and they may be used to implement the above system's requirements. Mondrian Memory Protection (MMP) allows a process to classify a single AS into multiple protection domains, and place word-granular permissions in each protection domain [66]. During runtime, threads use one of the protection domains and may switch between them. CHERI [67] provides byte-level memory protections with a capability model. A pointer in CHERI is a capability that includes both a memory address pointer and access permissions to that pointer. Greathouse et al. propose a very flexible, per-thread memory protection scheme using unlimited watchpoints [38]. A watchpoint specifies a particular memory location and it interrupts a thread's execution whenever that memory location is accessed. In their design, watchpoints have a flexible memory addressing scheme and can thus support both coarse and fine-grained memory locations. Although these seem particularly suited for our purposes, their downside is that they require new hardware primitives, which are not available in existing commodity hardware.

Table 2.1: General API of parallel memory permissions.

Syscall	**Description**
`mprotect_b(addr, prot)`	In dom_B, set the protections specified in `prot` on the virtual page with the address `addr`.
`switch_domain(dom)`	Switch the calling thread's protection domain.

Summary

The approaches described above either require non-trivial modifications to applications, impede parallelism, or depend on non-general hardware features. In this work, we designed parallel memory permissions so that it is not similarly restricted. It provides two protection domains, dom_A and dom_B, over a single AS. A system can protect the same virtual page differently in each domain, and switch threads between the two during runtime. In Table 2.1, we show the API of parallel memory permissions and in the following, we present two different ways of implementing it.

2.1 Multiple Page Tables

In modern systems that support virtual memory, the kernel allocates a single set of page tables per process. These tables map virtual addresses in an AS to their physical addresses in RAM, and they store the permission bits for each page. A straightforward way of implementing parallel memory permissions then is to have multiple sets of page tables per process. The kernel should synchronize their virtual-to-physical mappings while allowing the application to set different permission bits for the same virtual page, and it should allow threads to select which table to use when accessing the AS. We implemented the API in Table 2.1 with two sets of page tables per process, tbl_A and tbl_B; tbl_A is the page table already allocated by the kernel, while tbl_B is the new page table we introduced. They correspond to the protection domains dom_A and dom_B, respectively. In the following, we discuss aspects of their implementation in greater detail.

2.1.1 Synchronizing virtual-to-physical mappings

Since tbl_A and tbl_B are meant for the same AS, they need to maintain the same virtual-to-physical mappings. At first glance, this appears difficult; threads running in parallel may modify the shared AS (e.g., by mapping new pages) while using either of the two tables. In such situations, the application should work as expected and we may not impose on them undue requirements (e.g., only modify the shared AS when all threads use the same set of page tables). We make this task tractable by taking advantage of the design of modern OSes.

Consider the Linux kernel. It manages a process's AS with `mm_struct`, an architecture-neutral data structure. The `mm_struct` maintains the metadata of an AS and the kernel uses them to perform its virtual memory tasks, such as deciding when to perform a copy-on-write. The kernel manages page tables using a four-level hierarchy, starting from the `pgd` — Page Global Directory — and it keeps a single pointer to the `pgd` in `mm_struct`. When the kernel needs to update the page table of an AS, it does so using a series of routines that operate on the `pgd`. These routines access and modify the remaining levels (`pud` — Page Upper Directory, `pmd` — Page Middle Directory, and `pte` — Page Table Entries).

Although the kernel uses a four-level hierarchy, the CPU may use a different page table layout. Linux handles this potential discrepancy by leaving it to architecture-specific code to define the page table routines. For example, ARM CPUs using 32-bit virtual addresses keep a two-level page table. The kernel code that supports these particular CPUs fold the first three levels of the kernel's page table (`pgd`, `pud`, and `pmd`) into one to represent the first level table and uses `pte` to represent the second-level table.

In other words, Linux keeps a separation between the metadata about a process's AS (which is architecture-neutral) and the underlying CPU's page tables (which is

architecture-specific). We make use of this abstraction to implement parallel memory permissions. First, we add `pgd_B`, a new page table pointer that points to tbl_B, to `mm_struct`. Next, we modified the architecture-specific routines that modify the `pgd` (and the subsequent levels) to make the same changes to `pgd_B`. Finally, we reuse the same synchronization primitives the kernel already uses to make changes to the shared AS thread-safe.

Taken as a whole, our approach enables tbl_A and tbl_B to be synchronized in a thread-safe manner without requiring excessively invasive changes to the kernel. For instance, we did not need to duplicate and synchronize the AS metadata in `mm_struct`. It would have taken more work to implement parallel memory permission if the kernel tightly coupled the AS's metadata and the CPU's page tables. We briefly investigated the FreeBSD kernel and found it to have a similar abstraction. Therefore, we are confident that parallel memory permissions may be implemented in FreeBSD and leave it to future work to do so.

2.1.2 Managing the TLB

A CPU core typically caches the virtual-to-physical mappings of recently accessed virtual addresses in the Translation Lookaside Buffer (TLB). This reduces the need to repeatedly walk the page tables to resolve the physical address for a virtual address. The TLB may also contain other information, depending on the architecture. In ARMv7 (ARM CPUs running in 32-bit mode), the TLB includes the permissions for the virtual page, and each entry is tagged with an Address Space Identifier (ASID). The ASID is used by the core to distinguish mappings between processes. By default, the kernel assigns a single ASID for each process, which is insufficient when a process uses tbl_A and tbl_B.

Suppose a thread in a process P with ASID i accesses the virtual page V_x while using tbl_A. The TLB will cache V_x's virtual-to-physical mapping and permissions,

and it will tag the entry with i. If the thread subsequently switches to tbl_B and accesses V_x again, the core will use the cached entry in the TLB instead of walking through tbl_B. Hence, when it checks if the thread has access to V_x, it will rely on tbl_A's cached permissions instead of using those in tbl_B.

One way to address this is to simply flush the TLB whenever the thread switches between tbl_A and tbl_B. However, this will increase overhead and it might not be sufficient if the TLB is shared by multiple cores. Instead, we assign separate ASIDs, one for tbl_A and one for tbl_B.

2.1.3 Minimizing the page table size

We applied two different optimizations to reduce the memory consumption of the additional page tables. First, we lazily allocate `pgd_B`, when a process either switches to or sets protections in dom_B (i.e., when a process calls the API in Table 2.1). This minimizes the overhead of our kernel modifications in processes that do not use parallel memory permissions. Second, we try to share page tables between tbl_A and tbl_B where possible.

In the two-level page table layout used by ARMv7, a page's permission bits are kept in its corresponding entry in the second level table. Since tbl_A and tbl_B may keep the same permission bits for much of the shared AS, we allocate new second level tables for tbl_B on-demand. We illustrate this process in Figure 2.1. Here, since dom_A and dom_B have different permissions for V_x, tbl_A and tbl_B use separate second level tables to store the different permission bits. However, they share the second level table for V_y since its permissions are the same in the two protection domains.

2.2 ARMv7 memory domains

Even though parallel memory permissions does not require hardware primitives, its implementation is flexible and it may use any additional memory protection features

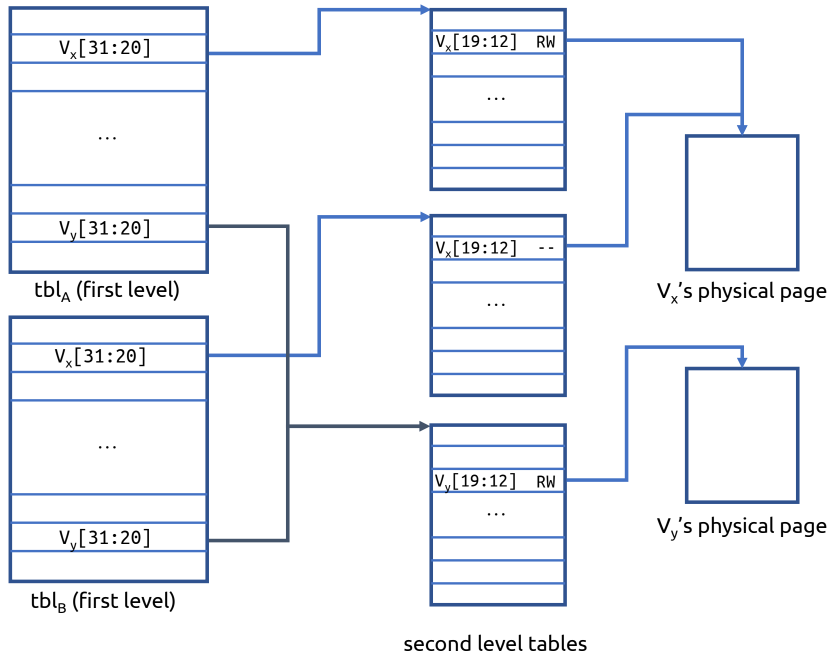

FIGURE 2.1: An illustration of how tbl_A and tbl_B share second-level page tables where possible, so as to reduce the total size occupied by the page tables.

the architecture provides. For example, ARMv7 provides a *memory domains* feature. Aside from the page protections in the second level table, each entry in the first level table can be tagged with one of 16 domains. In addition, each CPU core has a Domain Access Control Register (DACR) that specifies how accesses to tagged memory should be handled. The value of the DACR is a 32-bit, 16-entry vector specifying whether the core is in `no-access`, `manager`, or `client` mode with respect to each of the 16 memory domains.

If a core is in `client` mode for a domain, then attempts to access memory tagged under that domain are handled according to page protections. If a core is in `no-access` or `manager` mode for a domain, attempts to access memory tagged under that domain are always denied or allowed, respectively, and page protections are ignored. We use the following scheme to implement parallel memory permissions

using ARMv7's memory domains.

When a page V_x is protected in dom_B, the page permission bits for V_x are set appropriately and the entry in the first level table corresponding to V_x is tagged with a domain d. Threads running in protection domain dom_A are given access to V_x by assigning them `manager` permissions over d. If threads switch to dom_B, their permissions to domain d are downgraded to `client`. As a result, threads in dom_A have unconditional access to V_x while in dom_B, they abide by page protections.

The main limitation with memory domains is that the `no-access` and `manager` access modes apply to every page in a 1 MB region (since domains are tagged in the first level table and the second level table contains 256 entries, each pointing to a 4 KB page). The benefit however is that the kernel does not have to maintain and synchronize multiple page tables. We leave it to systems that require parallel memory permissions to decide if the tradeoffs are appropriate given their requirements.

2.2.1 Managing copy-on-write

The key challenge with using memory domains is that it may circumvent the kernel's use of copy-on-write. For example, when a process first maps a page at V_x, the kernel defers the work of allocating a new physical page. Instead, it sets write protections on V_x and updates the page table so that accesses to V_x go to a shared physical zero page. When the process attempts to write to V_x, the core will raise a trap. The kernel relies on this trap to allocate a new physical page for V_x, and to update its virtual-to-physical mapping.

Suppose V_x falls under the domain d while it points to the zero page. If a thread writes to V_x while having `manager` permissions over d, the write access will succeed and the thread will modify the zero physical page. This is dangerous and it typically causes a system-wide crash, since the zero page is shared across all processes and is always expected to be fully zeroed.

We handle this by essentially disabling copy-on-write for pages that fall in domain d. When a first level table entry is initially tagged with d, we loop through the second level table it points to, and allocate new physical pages for the virtual pages that are marked copy-on-write. Next, whenever the kernel happens to map a new page in d, we immediately allocate the physical page instead of deferring it. The consequence of this approach is that in the worst case, we would have allocated 1 MB worth of pages.

2.3 Tradeoffs

Compared to some of the existing work presented earlier, such as Mondrian Memory Protections [66], CHERI [67], and Unlimited Watchpoints [38], the protections offered by parallel memory permissions are limited. It only has two protection domains (dom_A and dom_B), and the protections are page-granular (as opposed to word- or byte-granular). The tradeoff, however, is that it does not require any special hardware primitives to function. It works in any architecture that provides virtual memory using page tables, supports parallel execution of threads, and does not require significant modifications to applications to use them (if at all). As we demonstrate in Chapter 3 and Chapter 4, the present implementation of parallel permissions is powerful even with these limitations. In Section 4.4.4, we discuss how these limitations may be mitigated.

3

SandTrap: Tracking Information Flows On Demand with Parallel Permissions

3.1 Introduction

Dynamic information-flow tracking (DIFT or taint tracking) maintains data dependency information at runtime. TaintDroid [33] provides DIFT for managed code on Android, and has proved to be a valuable building block for numerous mobile services [28, 34, 36, 58, 59, 63]. Despite its popularity, TaintDroid does not precisely track flows through apps' native libraries. Native libraries are crucial for many modern apps that utilize sensor processing, computer vision, and augmented reality. As more app functionality moves to native code, the need to track flows through native libraries will grow. The primary challenge of tracking native code is that emulation overhead can be as large as 30x. We are unaware of any DIFT system that provides a practical balance of performance and precision for managed and native execution environments.

To reduce emulation overhead, prior work on *on-demand DIFT* [39, 53] aimed to only emulate threads while they handle tainted data; threads that never access

tainted data run at full speed. This approach typically relies on page protections to force untracked threads to trap when accessing a page holding tainted data. As a result, untracked threads run with memory protections enabled so that they trap appropriately, but tracked threads must run with protections disabled so that the emulation layer can access tainted data.

The main limitation of this prior work is that memory protections must be the same for all threads executing in parallel, which prevents tracked and untracked threads from running at the same time. This limitation is a poor fit for Android apps, which typically consist of a main UI thread running managed code and background threads that perform computationally intensive tasks using native libraries. Background threads often communicate with the main UI thread through shared buffers, and the main thread's managed runtime (e.g., Dalvik) needs unprotected access to those shared buffers so that the runtime itself does not trigger emulation. Thus, untracked background threads (i.e., the threads that would most benefit from on-demand DIFT) cannot run in parallel with the main UI thread.

In this chapter, we present the design, implementation, and evaluation of an on-demand DIFT system for Android called *SandTrap*. The key insight underlying the design of SandTrap is that *parallel memory permissions* are a crucial requirement for on-demand DIFT on mobile. In a previous position paper [55], we speculated that ARM memory domains could be a useful mechanism for making native DIFT overhead proportional to the amount of tainted data that native code processes. In this work, we present a complete system that achieves this goal using parallel memory permissions implemented with both domains and multiple page tables. We make the following contributions:

- We identify parallel memory permissions as a key requirement for on-demand native DIFT.

- SandTrap is the first system to support on-demand DIFT of multi-threaded applications without constraining parallelism.

- Using our SandTrap implementation to track microphone data through Instagram, we show that SandTrap's native execution is nearly seven times faster than continuous DIFT. For the same workload, SandTrap consumes only 10% more energy than stock Android, while continuous native DIFT consumes 44% more energy than stock Android.

3.2 Background

In this section, we provide background information on Android and dynamic information-flow tracking (DIFT).

3.2.1 Android: threads, native code, and shared buffers

The Android platform primarily consists of a Linux kernel, a runtime environment and support libraries for Dex bytecodes, and an inter-process communication system called Binder. Developers write most app code in Java, which is compiled into Dex bytecodes and distributed to users in `.dex` files. Apps can also include third-party native libraries, typically written in C or C++, that interact with bytecodes through the Java Native Interface (JNI). Android launches each app by forking a common zygote process and uses the managed runtime to load app-specific code instead of invoking the `execve()` system call. Each process runs under a per-app `UID` assigned by the kernel. Prior to Android 4.4, Android executed bytecodes in a Dalvik virtual machine and used just-in-time (JIT) compilation to improve performance. Starting with Android 4.4, Android introduced a new runtime environment called the Android Runtime (ART). ART compiles most bytecodes ahead-of-time (AOT) into a native ELF executable when an app is installed.

Nearly all Android apps are multi-threaded, with one main UI thread that is responsible for quickly responding to user inputs, and background threads responsible for slow tasks such as network communication and database queries. Android enforces this division of responsibilities: attempts to access the network from the main thread trigger an exception, and slow event processing on the main thread causes an "App Not Responding" (ANR) dialog. In addition to I/O requests, developers often offload computationally intensive tasks such as image processing and machine-learning classification to native libraries running on background threads. The JNI manages transitions between managed code (i.e., Dalvik or ART) and native code, and provides a set of methods for sharing data between the two execution modes.

While native code indirectly accesses most Java object state through JNI methods, they may obtain pointers to the raw memory backing some Java objects. First, for each native Java type, such as `int` and `char`, native code can directly access the backing memory for a Java array of that type using the `GetArrayElements()` family of methods. The JNI will either return a pointer to the backing memory of the Java array object or a copy of the object's backing memory, depending on the state of the garbage collector. A call to `GetArrayElements()` must be accompanied by a call to `ReleaseArrayElements()` so that data can be copied back (if necessary) and to notify the garbage collector that native code no longer needs the object. Second, direct `ByteBuffers` allow native and managed code to share access to large, long-lived regions of memory, such as buffers holding image data. Finally, native code can access a Java string's backing memory using the `GetStringChars()` family of methods. These calls must be accompanied by calls to `ReleaseStringChars()`. For many apps, shared buffers are critical for good native performance since they reduce the number of JNI method calls needed to access a Java object, and they eliminate copying between multiple representations of the same object.

To illustrate the interplay of background threads, native code, and shared buffers

on Android, we will briefly describe how the built-in camera app saves a `JPEG` image. When a user takes a picture, the app receives camera data in `YCbCr` format. To convert the image from `YCbCr` to `JPEG` format, the camera app calls into a native library and passes references to four direct `ByteBuffers`: three for the input image (`Y` channel, `Cb` channel, and `Cr` channel) and one for the output `JPEG` image. Native code compresses the input image, stores the result in the output buffer, and then returns to managed code, which can display the resulting `JPEG` image or save it to a file.

While this is one example of how background threads, native code, and shared buffers can be used, we have observed similar patterns in apps that process audio and image data, perform encryption and decryption, perform machine-learning classification, and render game content.

3.2.2 DIFT: performance and precision

Dynamic information-flow tracking (DIFT or taint-tracking) records data dependencies as a process executes. DIFT systems maintain a *label*, often a bit vector, for each information-holding object, such as a file or program variable. At any moment, an object's label reflects whether it contains information derived from a set of tracked *sources*, such as a file, network socket, or sensor. As processes perform *operations* on data objects they transfer information between objects, and DIFT systems dynamically update destinations' labels according to a propagation logic.

DIFT systems can track information flows at many granularities, and system designers must trade off tracking precision for overhead and developer burden. Precise tracking requires expensive hardware emulation, typically by interposing on individual machine instructions. Greater precision also requires more label storage, often one label for every machine register and 32-bit range of valid memory. However, because precise DIFT maintains more detailed information about the locations of

tracked data, it can be transparently applied to unmodified executables without inducing high false-positive rates.

Imprecise DIFT is faster and requires less label storage than precise DIFT, but requires a set of trusted *declassifiers* to avoid false positives [31, 44, 69]. Declassifiers are trusted to clear bits from objects' labels, but integrating declassifiers into existing code bases requires developers to refactor applications into trusted and untrusted components. Declassifiers can also be difficult to write and reason about because of the limited information available to them when making declassification decisions.

Performance and precision are both important for DIFT, and TaintDroid [33] is a widely used DIFT system for Android that provides a good balance of the two by tracking information through program variables within the Dalvik runtime. On microbenchmarks, TaintDroid DIFT imposes only 14% overhead. TaintDroid's appealing combination of performance and precision has made it a core building block of numerous higher-level services, including trustworthy sensing [36], data deletion [59], cache eviction [63], energy conservation [58], and password management [28]. TaintDroid has also been instrumental for studying how mobile apps handle sensitive information [33, 34].

Despite its widespread use, TaintDroid does not precisely track flows through native code. TaintDroid does not even allow apps to load third-party native libraries, and it provides imprecise, method-level tracking for platform native libraries. Disallowing third-party native libraries is an increasingly problematic limitation of TaintDroid. We collected information from 86,000 apps in the Google Play Store in March of 2016, each with more than 100,000 downloads, and found that over 43% used at least one third-party native library. This is a large increase from the 5% reported in the TaintDroid paper from 2010. Furthermore, we have observed many flows through third-party native code that might be useful to track. For example, Instagram stores image thumbnails using native libraries, and Sony's TrackID app identifies songs by

forwarding audio-stream fingerprints to a remote server using native libraries.

The main barrier to precisely tracking native third-party libraries is poor performance. Precise DIFT for machine code like native libraries is slow because the ratio of tracking-instructions to application-instructions is much higher than in a managed runtime like ART or Dalvik. Executing a typical Dex bytecode requires executing tens of machine instructions to parse the bytecode source and destination registers, perform basic sanity checks, set the values of the destination registers, and then move on to the next bytecode. In TaintDroid, performing DIFT on a bytecode requires just a few additional instructions: loads for the source objects' labels, a bitwise OR for the propagation logic, and stores to the destination objects' labels. As with TaintDroid, DIFT for ART can be fast and precise. TaintART's integration of DIFT into ART exhibits a runtime overhead of only 14% [61].

The low overhead of performing DIFT in ART and Dalvik is in stark contrast to the high overhead of performing DIFT on native code. Droidscope [68] and NDroid [52] both provide DIFT for Android across managed and native code through an x86-QEMU virtual machine, and exhibit slowdowns of between 12x and 34x compared to an x86-QEMU baseline. These systems cannot run on typical mobile hardware, and their performance results are consistent with earlier work showing native DIFT overheads between 10x and 30x [20, 50].

Over the last ten years, many projects have proposed techniques for improving native DIFT performance, but none are a good match for Android. First, static analysis can improve tracking performance, but this approach is prone to false positives due to aliasing and context sensitivity [32, 43]. These issues are particularly challenging with native code, and the popular static information-flow tracking tool FlowDroid [10] does not support native code.

Minemu [15] achieves 1.5x to 3x slowdowns on x86 in part by repurposing unused SSE registers for label storage and using SIMD instructions for propagation.

However, ARM's SIMD/NEON extensions are critical to many native libraries for Android. It is worth noting that to the best of our knowledge no prior DIFT system supports SIMD/NEON instructions, including Droidscope and NDroid. More recently, JetStream [54] accelerated DIFT for replayed processes by parallelizing the work across a compute cluster, but this approach is limited to replayed processes.

The most promising approach to improving native DIFT performance on Android is to track on-demand by emulating only the instructions that handle tracked data [39, 53, 55]. This can significantly improve native DIFT performance when the app's code spends a small portion of its time processing tracked data. For example, if a system tracks location or microphone data, then native libraries that perform image processing should not be tracked.

Xen [39] uses page protections to track on-demand by placing read and write protections on pages holding tracked data. As long as a VM processes untracked data, it runs at normal speeds. Accessing a protected page triggers a page fault that forces the VM to enter emulation. Only one set of protections can be active at any moment, and the hypervisor removes DIFT-related protections from tracked pages during emulation so that instrumented code can read and write tracked data and label storage without triggering further faults. If an emulated VM runs long enough without tracked data in its registers, the hypervisor can stop tracking the VM by restoring page protections and running the VM at full speed.

LIFT [53] avoids unnecessary emulation by performing live-in and live-out analysis of code segments to identify when a segment can run at full speed. Segments with untracked inputs and outputs do not need to be emulated, and any segment that could either read or overwrite tracked data must be emulated. Like Xen, LIFT uses page protections to prevent untracked code from accessing label storage and removes those protections during emulation. As a result, neither Xen nor LIFT can run threads in parallel.

More generally, requiring different protections for tracked and untracked threads, while using a protection model that allows only one set of active protections, will limit parallelism; tracked threads cannot run with protections enabled, and untracked threads cannot trap with protections disabled. On a single-core machine, systems like LIFT or Xen could restore the appropriate protections when switching the processor between tracked and untracked threads. For multicore processors, untracked threads cannot run as long as threads are tracked on other cores because only one set of protections can be active at a time.

If a DIFT system runs on a multicore system and must rely on the default page protections behavior, it must do one of two things when a thread in a process accesses tracked data and begins emulation: (i) emulate all threads in the process, so that future accesses to tracked data by currently untracked threads are identified and tracked by the system, or (ii) use a permissions-driven execution model so that threads with different page protection requirements do not execute in parallel. The first is akin to continuous DIFT and defeats the purpose of performing on-demand tracking. The second limits parallelism.

We demonstrate the performance impact of the latter approach with an experiment using eBooka, an ebook reader for Android. When a user opens a PDF document, the app displays a circular loading bar in the UI while using native code in background threads to load the file and prepare it for rendering. Once the file is loaded, the app displays the first page of the document and hides the loading bar.

We begin our experiment by continuously recording the frame rate of the UI. Next, we wait for 10 seconds before interacting with the app, giving it time to load a complex PDF file. Finally, we wait for one minute before terminating the experiment. We ran this on a Galaxy Nexus smartphone, which has a dual-core CPU, using parallel and permissions-driven execution. To enforce the latter, we modified the Linux scheduler to ensure that threads within a process are not scheduled in parallel

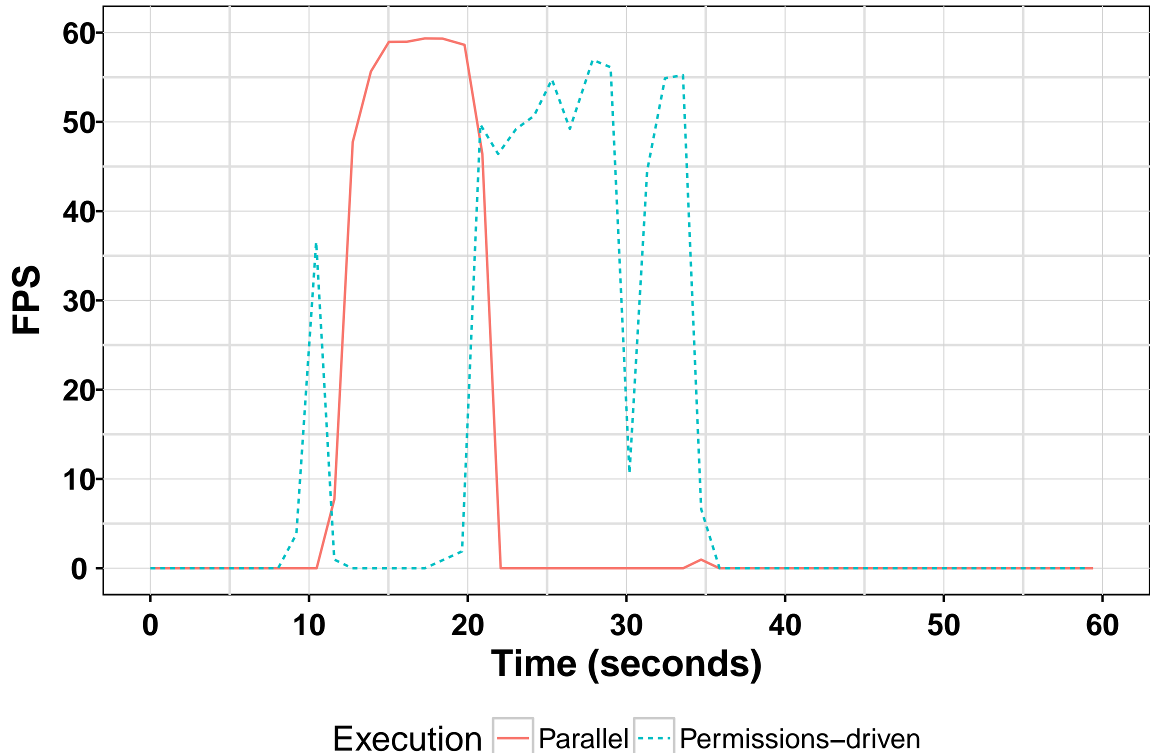

FIGURE 3.1: The frame rate recorded from a dual-core Android smartphone when loading a PDF file in eBooka, an ebook reader app, with parallel and permissions-driven execution.

if they require different page protections. Our results are in Figure 3.1.

In both cases, the frame rate starts out at 0 FPS due to an Android optimization in which the renderer simply displays the previous frame instead of rendering a new frame when there are no UI updates. When all threads can run in parallel, the frame rate jumps to 60 FPS after the initial wait period. This is when the app displays a "loading" message and the loading circle continuously spins. At time $t = 22$, the file is loaded and the first page is displayed. As there are no more updates to the UI, the frame rate drops back to 0 FPS.

Under permissions-driven execution, the main UI thread (running managed code) cannot run in parallel with untracked threads because it requires unprotected access to pages while untracked threads require protected access (see Section 3.3.1). This

will either cause the UI to be unresponsive (when untracked threads run) or prolong the time taken by untracked threads to complete their tasks (when the UI thread runs). Both effects are observable in Figure 3.1. After the initial wait period, the app displays the loading bar just before the frame rate drops to 0 FPS for about 8 seconds. During this interval, the UI thread does not run because an untracked background thread is running. Any input events from the user during this period would not be handled within Android's 5-second deadline, which will cause the ANR dialog to appear [1]. Between $t = 20$ and $t = 33$, the UI stutters because there is contention between the UI thread and the background thread. Eventually, at $t = 35$, the file is loaded and the frame rate drops back to 0 FPS as there are no more updates to the UI. As expected, permissions-driven execution limit parallelism and induce poor performance and interactivity.

To summarize, we are unaware of any system that meets our goal of providing practical DIFT for Android across managed and native code. Any solution should support unmodified apps and make performance overhead proportional to the amount of tracked data that third-party native code processes.

3.3 System Overview

In this section, we provide an overview of SandTrap, including the principles that guided its design and our trust-and-threat model.

3.3.1 Design principles

To provide practical DIFT on Android for managed and native code, we designed SandTrap using the following design principles.

Track managed and native code separately.

One approach of tracking managed code and native libraries in Android is to use the same instruction-level tracking for both. DroidScope and NDroid come closest to this approach by running the entire Android platform in an x86-QEMU virtual machine. Integrating DIFT into the QEMU hypervisor is appealing because it does not require platform changes and naturally tracks flows through managed and native code.

One problem with applying a single low-level DIFT mechanism to all execution environments is that understanding managed-code behavior is difficult. Because of this, DroidScope uses virtual-machine introspection to apply different levels of QEMU instrumentation to the Dalvik runtime and native libraries. This allows DroidScope to track all flows and reconstruct Dalvik-level program context, but it suffers from a 30x performance slowdown. NDroid improved the performance of DroidScope by integrating TaintDroid into the guest virtual machine rather than tracking managed code through QEMU. NDroid's overhead is an improved 12x, but this is still much higher than TaintDroid's and TaintArt's 14%.

As a result, SandTrap tracks managed code and third-party native libraries separately, relying on TaintDroid to continuously track managed code and using on-demand DIFT for third-party native code. Even though TaintDroid does not support the most recent versions of Android that use ART, TaintDroid's code is stable and widely used. Replacing TaintDroid with TaintART or another DIFT system for ART would require some engineering effort, but we do not anticipate it fundamentally changing our design or results.

TaintDroid tracks flows for each Dex bytecode that executes, whether it is interpreted or JIT compiled. TaintDroid uses 32-bit labels and in-lines labels with each object-instance's fields on the heap and with local variables on the stack. It

also maintains one label for each Java array object and each local register (including method parameters). Because of TaintDroid's relatively low overhead, SandTrap always performs DIFT on managed code.

For native DIFT, SandTrap targets 32-bit ARMv7 processors and maintains 32-bit labels that mirror the format of TaintDroid's labels. SandTrap maintains a label for each 32-bit word of memory, each 32-bit core register (excluding the PC), and each 64-bit extended register. SandTrap relies on MAMBO [37] to instrument third-party native libraries. Like MAMBO, SandTrap does not support deprecated Jazelle and thumbEE instructions, but supports the remainder of the ARMv7 ISA, including floating-point, SIMD/NEON, and thumb instructions. To the best of our knowledge, SandTrap is the first implementation of DIFT for ARMv7 to support all non-deprecated portions of the ARMv7 ISA. SandTrap also monitors the Dalvik JNI layer to manage transitions between managed code and native code. The primary responsibility of this code is synchronizing labels between Dalvik and native representations.

Given this design, a SandTrap thread can exist in one of three states: *managed*, *tracked*, and *untracked*. A managed thread executes under TaintDroid. A tracked thread executes third-party native code under native DIFT using MAMBO instrumentation. An untracked thread executes third-party native code without MAMBO instrumentation. As we will see, the main technical challenge for SandTrap is ensuring that an arbitrary mix of managed, tracked, and untracked threads can execute in parallel.

Parallel over permissions-driven execution.

We initially thought that tracking managed code and native code separately would mitigate most of the problems with permissions-driven execution, since managed runtimes provide a software-based protection mechanism that is orthogonal to hardware-

enforced page protections. As long as managed-runtime code and state reside on a set of pages that is *disjoint* from the set of pages holding native-library code and state, changing the protections of native pages will have no effect on the managed runtime.

Unfortunately, the widespread use of shared buffers across the JNI boundary in Android prevents managed threads from running in parallel with untracked threads under permissions-driven execution. To see why, consider a memory-protection scheme that supports only a single set of active protections for all running threads, and assume that native code obtains a pointer to a shared buffer via one of the JNI calls discussed in Section 3.2.1.

As long as accesses to shared buffers are properly synchronized (i.e., the program is race free) and all shared buffers reside on pages that are clear of tracked data, then managed and untracked threads can run in parallel. If a managed (or tracked) thread acquires exclusive access to the buffer (e.g., it acquires a lock) and copies tracked data into the shared buffer, then its exclusive access will also allow it to disable read and write access to the buffer's pages before an untracked thread can access it.

However, if at any moment a shared buffer resides on a page containing tracked data, then untracked threads must trap if they access the page. At the same time, tracked and managed threads should not trap when accessing it. Thus, under permissions-driven execution, untracked and managed threads cannot run in parallel when a shared buffer resides on a page containing tracked data. As demonstrated earlier, this restriction would cause poor interactivity and a possible ANR dialog because the main thread has to pause whenever untracked threads execute.

To avoid the potential for these dangerous stalls, SandTrap uses *parallel memory permissions*, which allows managed threads (especially the main thread) to safely run in parallel with tracked *and* untracked threads. An alternative to using parallel memory permissions is to map physical pages holding tracked data into two different

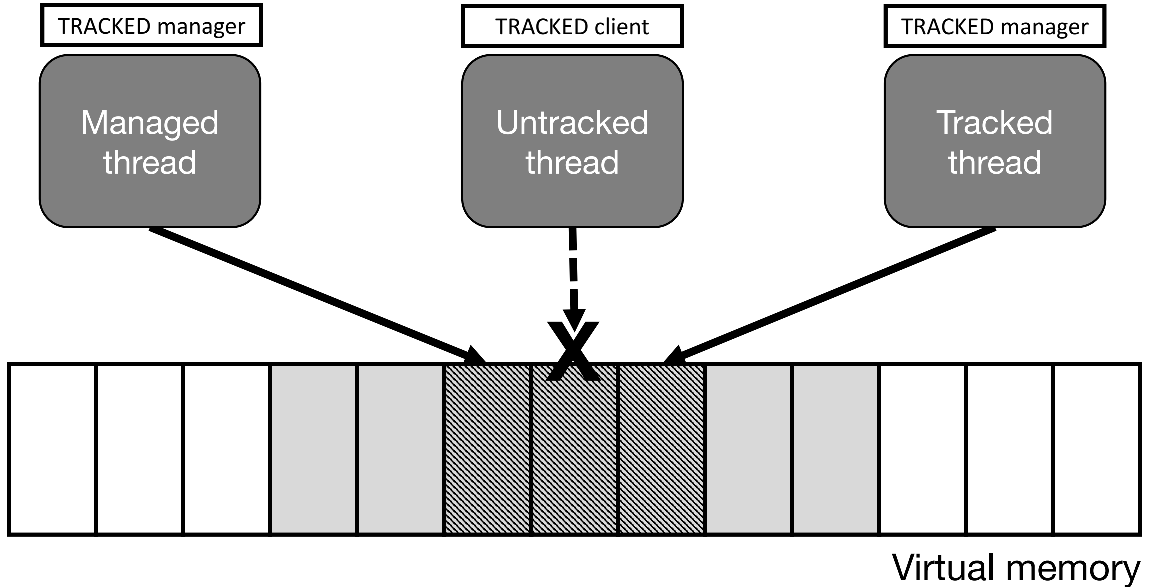

FIGURE 3.2: An illustration of how SandTrap uses parallel memory permissions with ARM memory domains.

virtual pages. Different permissions can be applied to the two virtual pages, and memory accesses can be redirected to either page depending on the thread's status [9]. The problem with this approach is that an untracked thread may obtain a pointer to a clean buffer at one point in time, and then dereference the pointer after the buffer holds tracked data. Inspecting each memory access so that it can be redirected to the proper virtual page would require continuous emulation, and we are unaware of any way to correctly (much less efficiently) search for and rewrite pointers stored in a process's state as it executes.

ARM Memory Domains: SandTrap uses the ARM memory domains implementation of parallel memory permission by tagging all native pages holding tracked data with a new `TRACKED` domain. Figure 3.2 shows how SandTrap protects and

tags regions of memory holding tracked data. When tracked data is written to a page, read and write protections are placed on the page so that untracked threads trap when the tracked data is accessed. Next, SandTrap tags the first level table entry corresponding to the page with the `TRACKED` memory domain. Managed and tracked threads run in `manager` mode for the `TRACKED` domain, so that they bypass any protections placed on tracked pages. At the same time, untracked threads run in `client` mode for the `TRACKED` domain and abide by the tracked pages' protections. This memory-protection scheme allows SandTrap to perform on-demand native DIFT while executing an arbitrary mix of managed, tracked, and untracked threads in parallel with only a single page table per app.

Two page tables per app: We use the multiple page tables implementation of parallel memory permissions to understand SandTrap's performance in the general case, when special hardware features are not used. We designate the two page tables as *unprotected* and *protected*, and the latter has read and write protections on pages with tracked data. The unprotected table is used by managed and tracked threads while the protected table is used by untracked threads.

Track some platform libraries imprecisely.

Android provides a number of native platform libraries for native and managed code. These libraries offer a range of functionality, including the Bionic standard-C library and stubs for accessing hardware accelerators. Precisely tracking flows through many of these libraries (as opposed to native third-party libraries) would lead to correctness problems, and as a result SandTrap imprecisely tracks three broad classes of platform native libraries.

First, SandTrap does not precisely track the Bionic methods that it uses. In many cases, emulating Bionic would lead to recursive emulation or deadlock. For example, the Bionic implementations of `malloc()` and `free()` use a global lock for synchro-

nization, and emulating calls to `malloc()` will cause a deadlock if SandTrap calls `malloc()` during emulation. Instead, SandTrap performs method-grained tracking on these calls using our understanding of the call semantics to appropriately update labels.

Second, SandTrap does not precisely track Bionic's I/O methods, such as file system and network reads and writes. Though SandTrap does not use these methods itself, we instrument rather than emulate them to identify when tracked data leaves an app. For example, in order to propagate labels to files, SandTrap code in Bionic's `write()` method updates the extended attributes of the target file. Similarly, if tracked data is sent over the network, SandTrap will log information about the remote socket and the output buffer's label.

Finally, Android provides native stubs for interacting with hardware accelerators like GPUs. SandTrap cannot precisely track code that executes off of the main CPU, and so it interposes on accelerator methods and updates output labels based on the specified inputs and outputs. For example, many apps use OpenGL to render scenes and perform image processing. The OpenGL API provides methods for specifying configuration parameters, input buffers, shaders, and output buffers. SandTrap monitors these calls, and once native code launches a task on the GPU, we set the output buffer's label to the union of all input labels.

3.3.2 Trust and threat model

SandTrap relies on numerous existing pieces of software, including Android, TaintDroid, and MAMBO, and it inherits their trust-and-threat models. For example, SandTrap does not handle implicit flows, although it could borrow techniques from SpanDex [28] to limit them. Furthermore, our SandTrap implementation does not prevent buggy or malicious native code from corrupting label storage or Dalvik state. However, this is not a fundamental limitation since SandTrap could protect labels

and Dalvik state using parallel permissions or integrate memory checks into MAMBO emulation. We leave these measures for future work.

3.4 SandTrap

SandTrap continuously tracks Dex bytecodes using TaintDroid and performs on-demand DIFT on third-party native libraries. A SandTrap thread can execute in one of three states: managed (under the control of TaintDroid), tracked (under the control of native DIFT emulation), and untracked (no emulation). SandTrap's primary goal is to use parallel memory permissions so that managed, tracked, and untracked threads can execute in parallel.

3.4.1 Label storage

The first challenge that SandTrap addresses is synchronizing native and TaintDroid labels. TaintDroid labels are 32-bit vectors, where each bit represents a different tracked source, such as GPS, microphone, and camera. SandTrap's native labels have the same format as TaintDroid's labels.

An app's primary stack region resides at the top of the user-accessible portion of its address space (AS). Native-label storage for the primary stack resides at the bottom of the AS, and SandTrap maps each word of the stack region to its label using a fixed offset.

All native code, including the Dalvik VM, TaintDroid, SandTrap, platform libraries, and third-party libraries reside in a read-only code region just above the primary stack's label storage. Each app's heap resides above the code region. The heap is partitioned into a Dalvik-managed range and a native-managed range. Dalvik manages its range via `mspace_malloc()`, whereas SandTrap, platform libraries, and third-party libraries collectively manage the remaining heap memory using `malloc()`. Dalvik stores all Java objects and managed-thread stacks in its portion of the heap.

As previously mentioned, TaintDroid in-lines labels for Java object fields and local variables.

Native labels for non-Dalvik heap words reside in a 300 MB region immediately below the process's primary stack region. SandTrap allocates stacks for new threads by mapping new memory above the heap region (with `mmap()`). Thread stacks never share pages. Labels for native thread stacks reside at the same fixed offset used to locate heap-word labels.

To synchronize native labels and TaintDroid labels, SandTrap interposes on transitions across the JNI and monitors requests for pointers to shared buffers by native code. When managed code invokes a third-party native method, it can pass Java native types and Java object references. References and native types are passed by copy, and SandTrap ensures that the TaintDroid labels corresponding to the arguments are also copied to the appropriate native labels. It performs similar bookkeeping for return values and indirect accesses to Java objects through the JNI.

Synchronizing native and TaintDroid labels for shared arrays and `ByteBuffers` is slightly different because native code directly accesses these buffers. Recall that TaintDroid maintains a single label for each array, whereas SandTrap maintains per-entry labels for native arrays. Thus, when native code obtains a pointer to a Java array via the `GetArrayElements()` family of methods, SandTrap copies the Java array label into the native label for each native-array entry. Also recall that calls to `GetArrayElements()` can return a copy of the memory backing the Java object and that these calls must be accompanied by a corresponding call to `ReleaseArray-Elements()`. Release calls may synchronize a copy array with the actual Java object. SandTrap uses these release calls to synchronize labels, and sets the Java array's label as the union of all native-array entry labels. SandTrap handles shared Java strings similarly.

Unlike Java strings and array objects, direct `ByteBuffers` are long-lived and do

not require native code to explicitly release their pointers. Android recommends that Java code access these objects via `get()` and `set()` methods rather than directly accessing the underlying byte array. Thus, when Dex bytecodes read from a `ByteBuffer` via `get()` methods, SandTrap sets the TaintDroid label for the method's return value equal to the corresponding native label. When Dex bytecodes write to a `ByteBuffer` via a `set()` method, SandTrap copies the TaintDroid label for the argument to the corresponding native label. As a result, even though TaintDroid maintains a single label for each array, SandTrap effectively provides per-entry labels for direct `ByteBuffers` through the `get()` and `set()` methods. Label synchronization between TaintDroid and SandTrap for direct `ByteBuffers` may be optimized by doing so only for the buffers native code has accessed via `GetDirectBufferAddress()`. We leave this optimization for future work.

Note that we assume that apps are race free. In particular, we assume that threads obtain exclusive access to shared objects when updating them and that label updates are protected by this same exclusive access.

3.4.2 Parallel memory permissions

SandTrap uses both implementations of parallel memory permissions discussed in Chapter 2. SandTrap maintains a count of the number of tracked words in each page that may be accessed by native code. If a page's count increases from zero to one, then the page must be protected and if the count decreases from one to zero, the page's protections must be removed. This ensures that untracked threads trap when they access tracked data.

When using ARM memory domains, our current implementation of SandTrap does not allow pages in the `TRACKED` domain to be shared with other processes. This is not a fundamental limitation; to correctly support shared pages, the SandTrap labels of a shared tracked page should also be shared. We leave this engineering for

future work. That said, we have not seen this behavior in practice, and Android apps typically use Binder for inter-app communication.

3.4.3 Thread transitions

The final challenge that SandTrap addresses is managing thread modes and transitions. SandTrap does not allow native code to create new threads, and every thread starts as a managed thread under the control of TaintDroid. When a managed thread invokes a third-party native method, SandTrap copies the method arguments and updates the native labels as described in Section 3.4.1. SandTrap then asks the kernel to change the state of the thread to untracked. Depending on the implementation of parallel memory permissions, the kernel either updates the DACR or the page table used.

Untracked threads run at native speeds as long as they do not access a page holding tracked data. If an untracked thread never accesses a tracked page, then when it returns from the entry method, SandTrap switches the state of the thread to managed before returning control to TaintDroid. However, if an untracked thread accesses a page with tracked data, it will cause a page fault and trigger DIFT emulation. SandTrap performs native DIFT within a fault handler running on the faulting thread's stack. In order to register a handler for each thread, we modified the Android implementation of `pthreads` to wrap the new threads' start methods with a hook method.

Inside the hook method, SandTrap registers a `SIGSEGV` fault handler. `SIGSEGV` is a synchronous signal, and the Linux kernel guarantees that if a thread with a registered `SIGSEGV` handler triggers a fault, the signal will be delivered to the faulting thread. This is critical for ensuring that SandTrap performs DIFT on the correct thread. Otherwise, the kernel could deliver the signal to an untracked thread that has not accessed tracked data, a tracked thread that is already running under DIFT,

or, worst of all, a managed thread like the main UI thread.

On a trap, SandTrap's signal handler switches the thread to tracked state. Next, it invokes MAMBO to start executing basic blocks beginning with the faulting instruction and using the thread's existing stack. As unmodified MAMBO expects to control a process from start to finish, we made extensive changes to MAMBO so that it can pause and resume emulation at arbitrary execution points. For example, some third-party native libraries invoke platform OpenGL routines during emulation. As mentioned previously, SandTrap must interpose on these libraries so that it can track flows in and out of the GPU. Therefore, when a tracked thread makes an OpenGL call, SandTrap pauses emulation, performs the OpenGL call, and then resumes emulation. We discuss some of these implementation details later in Section 3.5.

A consequence of SandTrap's memory-protection scheme is that each Dex call into third-party native code requires at least two kernel traps: one to change to untracked mode, and one to change back to managed mode on return. Upcalls from untracked threads into managed code also cause additional traps: one to change to managed mode, and one to change back to untracked mode. If an app does not amortize the cost of these boundary crossings, then its native calls may be slower under SandTrap, even if the calls never trigger emulation.

Finally, it may be possible for a thread to exit emulation once its registers are clear of tracked data. In our experience these moments are fleeting, and threads typically start processing tracked data again soon after their registers are clear. The overhead of thrashing between tracked and untracked states can be very high, since switching requires a kernel trap. As a result, SandTrap emulates tracked threads until they return control to TaintDroid.

3.5 Implementation

Our goal from the outset was to have a complete working system running on a smartphone device with unmodified Android apps downloaded from the Google Play Store. Realizing this goal turned out to be more challenging than we anticipated. As we developed our prototype, we found ourselves needing to address a number of important sub tasks, and we were constantly balancing having a working prototype against engineering effort and performance constraints. In this section, we discuss SandTrap's implementation in terms of its two main versions. In the first version, we incrementally built a working system capable of meeting our goal. Despite our attention to performance, we found that it was still excessively slow. In the second version, we identified the key causes of slowdowns and rebuilt those portions completely, thereby improving performance significantly.

3.5.1 The initial prototype

In this first version, we designed our prototype to emulate tracked threads with a *fetch* ⇒ *decode* ⇒ *propagate* ⇒ *execute* cycle.

Fetching instructions

During emulation, we fetch the next instruction to execute by indexing the memory location in the thread's `PC` register. When a thread traps, the `PC` initially points to the faulting instruction, which is the first instruction we emulate. At the end of the emulation cycle, we update the `PC` based on the current instruction's properties. If it is a branch, we update the `PC` to the new location. Otherwise, we increment the `PC` by the instruction's length. It is 4 bytes when the core is in `ARM` mode, and it is either 2 or 4 bytes in `Thumb` mode.

Decoding instructions

SandTrap needs to decode each instruction during emulation, so that it may perform information propagation based on the instruction's semantics. We initially used *darm* [42], a decoder written from the ground up to be efficient. However, as it did not support the full ARMv7 ISA, it could not decode a large number of instructions we encountered in practice with real apps. We also found it time-consuming to continuously add support for new instructions in darm. We eventually switched to *capstone* [2], which had complete support for the ISA. We also aggressively cached the decoded output of capstone, so that we needed to invoke it only when the emulator encounters new, undecoded, instructions. We explored a number of other optimizations, such as pre-decoding the popular instructions in the ISA, but we opted to leave them unimplemented until their need became clearer.

Propagating information

SandTrap maintains one label for each machine register and 32-bit word in memory. The propagation logic to copy the labels between the different storage areas depends on the type and semantics of the instruction. ARM uses a load-store architecture; instructions either operate exclusively on registers, or they access memory to load or store data from registers. In general, we require the exact register contents when propagating labels for memory-access instructions but not for register-access instructions.

Consider the register-access instruction `add r0, r1, r2` that adds the contents of `r1` and `r2`, and stores the result in `r0`. The propagation logic in this case is simply `Label(r0)` ← `Label(r1)` | `Label(r2)`, where | represents the bitwise OR operator. The exact values of `r1` and `r2` do not matter. However, consider the instruction `ldr r0, [r1]`. This loads a word from the memory location specified in `r1` and stores it into `r0`. The propagation logic for this instruction is `Label(r0)` ← `Label([r1]);`

we need to find the label for the memory address in `r1` and assign that to `r0`. Hence, the propagation logic requires the runtime value of `r1`.

We manually wrote the propagation logic for each instruction we encountered while testing different apps.

Executing instructions

Conceptually, this step is straightforward; the emulator has to execute each instruction so that the thread does its work. However, how tracked data is accessed plays a key role in how instructions are executed. Recall that as explained in Section 3.2.1, we may not turn off page protections or use permissions-driven execution. We initially relied on the `/proc/self/mem` interface provided by the kernel. A process may use this interface to access its own AS. It has to open the interface as it would a file, seek to a virtual address in its AS, and then read or write to the interface. Using this interface involves a trap to the kernel, which is not bound to user space page protections and is hence able to perform the required memory access on behalf of the user space emulator.

Since we access memory via `/proc/self/mem`, we are unable to have the CPU natively execute memory access instructions. Therefore, the initial version of our prototype executed instructions via a series of C routines that implement the logic of the different instructions. This allowed the prototype to redirect memory accesses to `/proc/self/mem` whenever necessary.

This approach enabled us to make some progress in writing a functional emulator. However, it had two major drawbacks. First, writing C routines to implement each instruction is not scalable. Second, trapping to the kernel to perform each memory access is too slow. In certain microbenchmarks, our prototype imposed slowdowns of over 10,000x. Consequently, we observed that it was crucial for memory accesses by the emulator to be as quick as native accesses.

Once we developed parallel memory permissions, there was no more need to redirect memory accesses during instruction execution. Hence, we used the following scheme to natively execute instructions instead of using our C routines. First, we copy a `(instruction, epilogue)` pair to a special page allocated by the emulator. Next, we execute a prologue sequence that saves the emulator's context, sets the emulated thread's context on the core, and points the `PC` to the start of the special page. The core will now execute the app's instruction, and then run the epilogue, which saves the thread's context and returns control to the emulator by restoring its context. The exceptions are branch instructions, which have to be manually executed so that the emulator keeps its control over the thread's control flow.

There are two performance issues in this scheme. First, the core maintains separate L1 instruction and data caches. After we copy the `(instruction, epilogue)` pair into the special page, we also need to clear the L1 instruction cache for the range of addresses occupied by the page. Otherwise, the core might execute the stale instructions in its cache. In ARMv7, we have to trap into the kernel to clear the L1 cache as it is a privileged operation. We circumvented this requirement by modifying the kernel and relying on an ARMv7 feature to disable the L1 cache for the special page.

The second issue with this scheme is the cost of the prologue and epilogue, relative to the single app instruction being executed. We optimized it by enqueuing as much instructions as possible in the special page before executing them all. This optimization is subject to some constraints. Consider the following sequence of instructions from the app: (R_1, R_2, M_3), where R and M are register and memory-access instructions, respectively. As explained earlier, when propagating labels for R_1 and R_2, the emulator does not need the contents of the registers. However, before it can propagate labels for M_3, it needs R_1 and R_2 to be executed because the registers used in M_3 might have data dependencies on R_1 and R_2. A similar situation arises

when instead of M_3, we encounter a branch instruction.

Note that as explained in Section 3.3.1, there are a number of routines that we track imprecisely. When these routines are encountered during emulation, they are executed directly instead of decoding their instructions. We implemented this by checking the value of the `PC` after executing a branch instruction. If it points to one of the imprecisely tracked routines, the emulator calls the routine directly.

3.5.2 Using MAMBO

While we were developing the first version of our prototype, we constantly tested it with apps such as Instagram to check its correctness. Despite some observed slowdowns, we found its performance promising and we were confident it could be optimized further. Once we completed the prototype, we did some in-depth evaluations and found that despite our attention to optimization, it was still unreasonably slow. For example, in some informal tests, it took Instagram about 20 minutes to process and post a picture taken with the camera; a task that otherwise takes in the order of tens of seconds. The performance appeared promising earlier only because we were testing Instagram with small images.

The key reason behind the slowdown is intuitive: to emulate each instruction in the app, we were executing in the order of thousands of instructions to perform the *fetch* $\Rightarrow$ *decode* $\Rightarrow$ *propagate* $\Rightarrow$ *execute* cycle. Despite the benefits of this approach, such as being able to precisely trace the state of a tracked thread during emulation, it became clear that it was inherently slow and another approach was required. As a result, we decided to incorporate MAMBO [37] into our prototype.

MAMBO is a dynamic binary instrumentation tool for ARM. It is modeled after PIN [49] and has low overhead. A key difference between MAMBO and the first version of our prototype is that it instruments code in terms of basic blocks (as opposed to single instructions). It also provides a plugin API, which allows the

addition of custom instructions during basic block construction. Using MAMBO in SandTrap meant modifying both MAMBO and large portions of our prototype.

Execution at arbitrary points

MAMBO is designed to run a process from start to finish. We modified it so that it can begin and end instrumentation at arbitrary points. When a thread first traps (due to a tracked data access), we use MAMBO to construct the first basic block starting from the faulting instruction. Next, we execute this block and as it runs, MAMBO will create new blocks as needed. We added some instrumentation in Dalvik so that MAMBO can recognize when a native method ends. We use this instrumentation to resume unmodified execution.

Instruction fetch and execution

For the most part, we leave it to MAMBO to fetch the required instructions, build basic blocks, and execute them. Our modifications to this process were mainly in supporting imprecisely tracked routines. MAMBO should not build basic blocks out of those routines. Instead, it should just call them directly, and resume the rest of the thread's execution.

Information propagation

The process of label propagation happens during basic block construction and execution. At a high-level, when MAMBO allocates a basic block B_x, we allocate a corresponding DIFT block D_x. This DIFT block consists of 4 byte words that encode how labels should be propagated for B_x. During runtime, after B_x executes, a DIFT handler reads D_x and performs the required label propagation.

We implemented this with MAMBO's plugin infrastructure. During basic block construction, MAMBO invokes our plugin each time it adds an instruction to B_x. The plugin decodes the instruction (using PIE [6], an efficient and complete ARM

encoder and decoder, made for and used by MAMBO) and then writes the appropriate propagation logic to D_x. If the instruction accesses memory, we also add some custom instructions to B_x, to copy the contents of the instruction's registers to D_x during runtime. Recall that as stated earlier, we require the registers' contents to perform label propagation for memory access instructions. Our plugin also adds instructions in B_x to call the DIFT handler at the end of the basic block.

With this approach, we incur the cost of decoding instructions and generating D_x just once for each basic block, when it is first constructed. Only the DIFT handler has to run each time the basic block runs. This is unlike the first version of our prototype, where the *fetch* ⇒ *decode* ⇒ *propagate* ⇒ *execute* cycle ran on a per-instruction basis. As a result, this second version of our prototype is significantly faster. Our experiments in Section 3.6.2 (Figure 3.5) suggest that it can be further improved by carefully optimizing the DIFT handler.

In summary, we used MAMBO and refactored key portions of our initial prototype. The result is an efficient DIFT emulator, capable of meeting our goal of running on a smartphone device, with unmodified apps. Our experience with building SandTrap has shown us that while correctness may not be compromised, we have to also aggressively pay attention to performance constraints. Otherwise, we may end up with a working but unreasonably slow prototype.

3.6 Evaluation

This section evaluates SandTrap. Our goal is to answer the following questions: 1) Does on-demand DIFT provide better performance than always tracking? 2) Does SandTrap incur false traps? If so, are they truly false traps, or do they eventually touch tracked data? 3) Is on-demand DIFT more energy-efficient than always tracking? To answer the first two questions, we provide data about only native-method performance, i.e., the results are not end-to-end. However, the energy data used to

answer the third question represents an end-to-end result.

3.6.1 *Experimental methodology*

We use a Galaxy Nexus, a dual-core smartphone, with Android 4.1.1 (Linux kernel v3.0.31) for all our experiments. To evaluate SandTrap we add instrumentation at certain points to obtain timing, basic block counts, etc. as needed depending on the specific experiment. A slightly modified stock Android serves as our baseline for comparison. For timing measurements, we modify Dalvik to log timestamps at third-party native method entry and exit. Requisite information is logged and retrieved using the Android Monitor infrastructure.

Table 3.1: Android applications used to test overall performance of SandTrap while tracking camera, location, and microphone data.

Application	**Description**	**Native Code Use**	**Experiment Input**
Nectroid	Streaming music player	MP3 decoding	Play song for 10 seconds
Anagram	Generates anagrams	Anagram generation	"mariecurie", "nitromagnesite", "recreationist"
920 Editor	Source code editor	Load files and search	Keyword search
NoteCipher	Note taking	Encryption	Decrypt, enter, save, then delete note
eBooka	ebook reader	Load file and render pages	Load ebook
QDict	Dictionary	Load and search dictionary	Lookup three phrases in "American Idioms"
APV PDF	PDF viewer	Load file, render pages	Load ARM v7 manual
Writeily	Markdown editor	Convert markdown to HTML	Convert a markdown document to HTML 5 times
MuPDF	PDF viewer	Load file, render pages	Load ARM v7 manual
Instagram	Social networking	Image capture and processing	Take and post photo online

We use ten Android applications for our evaluation that represent a variety of use cases. Table 3.1 lists our applications, how they use native code, and our experiment inputs. From previous work it is known that TaintDroid overhead is only 14%, therefore *we eliminate measuring Dalvik induced overhead and focus only on the overhead of native routines.* We track each native routine invocation, the time spent in each native routine, and the number of MAMBO basic blocks executed. We present aggregate results (i.e., the sum of the respective metric for all native method invocations) unless otherwise stated. We note that our performance results represent a worst case in overall execution time since we simply sum the time spent in each

native code invocation. Given state-of-the-art mobile platforms with multiple cores, some of our invocations would be able to run in parallel.

We use Instagram to explore, in detail, the various SandTrap overheads. There is some inherent noise in our experiments since we execute on a real system. For example, with Instagram we find that each experiment exhibits $200 \pm \epsilon$ native method invocations. Preliminary investigation of this variation indicates it is likely due to Instagram authentication activity. To standardize our Instagram analysis, we automate measurements using a Python script executing on a host platform (e.g., desktop) to issue commands to the phone over Android Debug Bridge (ADB). This enables launching apps, clicking on the UI, typing, etc., and collecting instrumentation output from the Android Monitor.

An important characteristic of Instagram is that it launches many background threads that process new images to apply various filters and generate thumbnails. Therefore, even after an image is posted on the web site, the phone could still be processing the original data in the background. Our experiments are designed to ensure that we capture all Instagram activity for a given post. We use our script to issue a series of commands for Instagram to take a photo from the front camera, annotate the image, and post it online.

3.6.2 On-demand vs continuous DIFT

For our experiments, we track three sources of sensitive information: camera, location, and microphone. We measure performance as the total time that all threads spend in native execution, normalized to a baseline of stock Android with several small modifications for timing instrumentation. Figure 3.3 shows SandTrap performance for ten applications using boxplots. We observe that SandTrap introduces modest slowdowns (1x to 8x). Instagram is the only app that accesses tracked data (camera), and it incurs the largest slowdown. For the other applications, the range

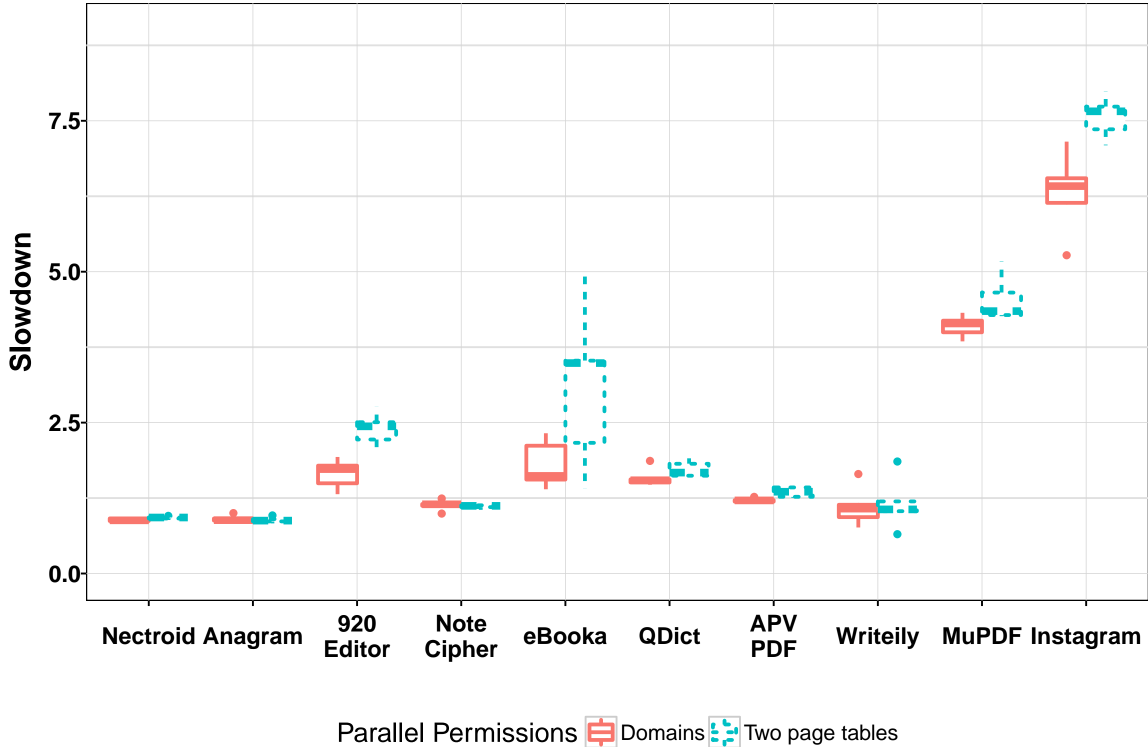

FIGURE 3.3: Overall slowdown imposed by SandTrap on native code execution in a variety of Android applications (tracked data sources: camera, location, and microphone).

of slowdowns is due to the varying number and duration of native calls. Recall that each native call in SandTrap incurs two traps, to switch between managed and untracked states. In some cases, such as MuPDF, the overhead is also due to the boundary crossings from native code to managed code. As discussed in Section 3.4.3, each upcall from native to managed code requires two additional traps. Observe too that, in general, SandTrap performs better with memory domains than with two page tables. This is also observable in the experiments below. We discuss these differences in more detail in Section 3.6.5.

We next examine SandTrap's overall performance using Instagram with differing information sources as a representative benchmark. We specify either the microphone or the camera as a taint source. These cases represent two extremes. Instagram

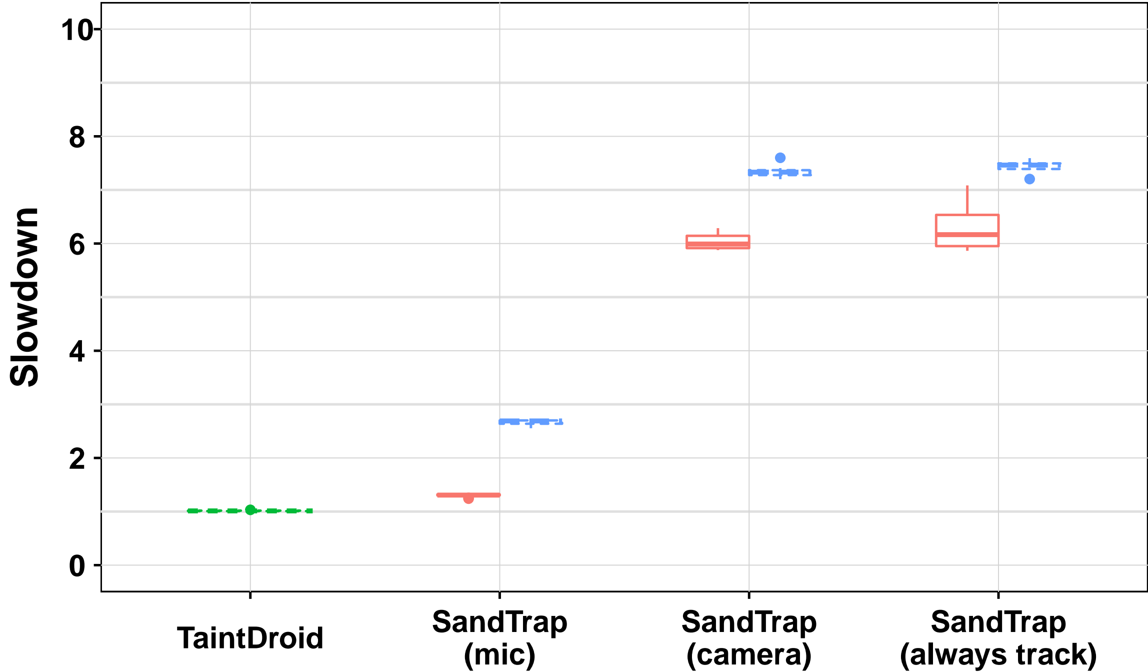

FIGURE 3.4: Slowdown imposed by SandTrap on native code execution while tracking different data sources through Instagram.

captures pictures with the camera and spends significant time in native routines processing the images. In contrast, when taking a photo, Instagram does not access the microphone, and as a result image-processing background threads should not incur much DIFT overhead. We also include results for unmodified TaintDroid and continuous DIFT.

The baseline stock Android spends approximately 10.5 seconds executing native code and has $200 \pm \epsilon$ native method invocations. The results, shown in Figure 3.4, reveal two important observations: 1) SandTrap performance approaches the baseline system for use cases that manipulate little or no tracked data, and 2) SandTrap incurs $\approx$6-8x slowdown when DIFT is required on most native routines. These results demonstrate the importance of on-demand DIFT and that SandTrap achieves worst-

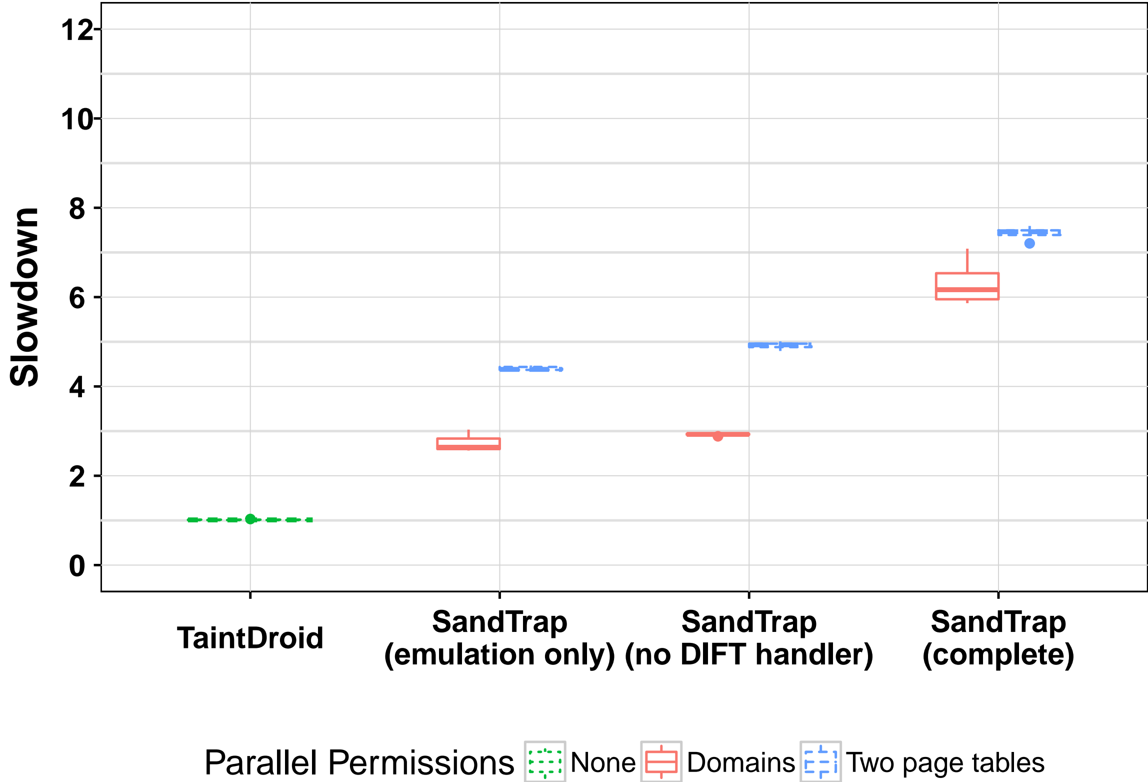

FIGURE 3.5: Individual SandTrap component overhead on native code execution in Instagram.

case performance comparable to other binary DIFT approaches.

TaintDroid achieves performance equal to stock Android, as expected since we focus only on native method execution and TaintDroid does not perform DIFT on native code. We observe that for the Instagram microphone source experiment, SandTrap incurs a modest 1.5x slowdown (using domains) due to the overhead involved in switching threads between the managed and untracked states. In contrast, always emulating and performing DIFT on native code incurs a 6-8x slowdown. SandTrap with the camera source incurs slowdown comparable to always performing DIFT on native code since Instagram performs substantial computation on image data.

To better understand SandTrap performance, we examine the overhead of individual SandTrap components. As discussed in Section 3.5.2, SandTrap uses MAMBO

to emulate a tracked thread. Beyond baseline MAMBO emulation, SandTrap adds two additional sources of overhead: 1) creating DIFT blocks, and 2) processing DIFT blocks with a DIFT handler at the end of each basic block. Here we examine the overhead for each of these components. To isolate the various SandTrap overheads, we examine the following three scenarios that emulates all native methods regardless of whether they access tracked data or not: 1) baseline MAMBO emulation, including trap overhead to begin emulation (emulation only), 2) DIFT block generation only (no DIFT handler), and 3) full tracking (complete). Each of these adds an incremental overhead beyond the baseline stock Android. Figure 3.5 shows the performance for Instagram using the camera source for TaintDroid and the three scenarios described above. The majority of overhead is in the DIFT handler that processes the DIFT block at the end of each MAMBO basic block. There may be opportunity to reduce this overhead since we have not optimized the DIFT handler in our implementation.

3.6.3 False traps

Since SandTrap uses coarse-grain page protections to trap accesses to tracked data, it will also trap native methods that 1) access non-tracked data that shares a page with tracked data, or 2) overwrite tracked data with non-tracked data. A false trap occurs if we begin emulation for a native method but it never actually processes tracked data before returning to Dalvik. In some cases, even though a trap may occur due to a native method accessing non-tracked data, it may still process tracked data eventually, which would cause a trap if we had fine-grain access control. In these situations, the native method simply incurred the trap earlier than strictly necessary.

We ran another set of our Instagram experiments (with two page tables) to analyze why the native methods traps. In sum, there were a total of 1,996 emulated native methods. Our results reveal that false traps are rare (12 invocations) and

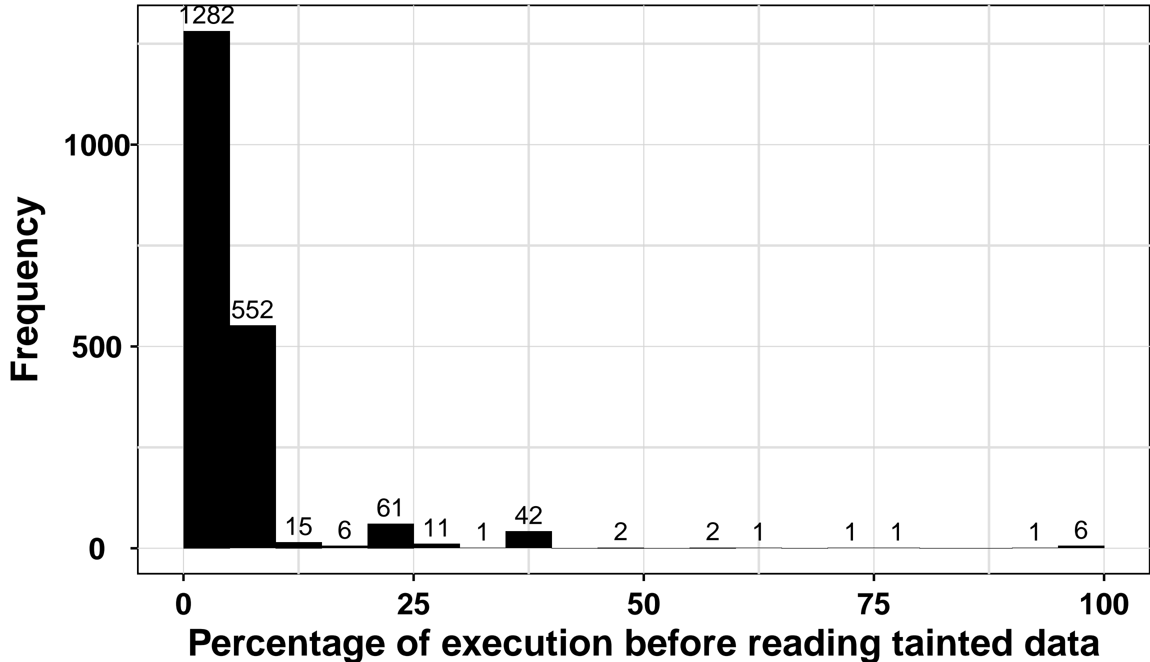

Figure 3.6: A measure of how quickly a native routine in Instagram reads tainted data after emulation begins.

only small percentage of methods read tracked data within the first MAMBO block executed (236 invocations). To better understand when a native method accesses tainted data, we count the number of dynamic MAMBO block executions after a trap until at least one register contains tracked data (first block). We also count the total number of dynamic MAMBO block executions after a trap until the native method exits (total blocks). The first block count divided by total blocks indicates, as a percentage, how far into the native method execution reaches before it accesses tracked data after a trap.

Figure 3.6 shows an aggregate histogram where x-axis is the percentage of execution for each native invocation in dynamic MAMBO blocks, where each bin is 5%. The y-axis is the count of instances that read tracked data for that given percentage of execution. From this figure we observe that most native invocations read tracked data within the first 5-10 percent of blocks after a trap. Specifically, 92% of native

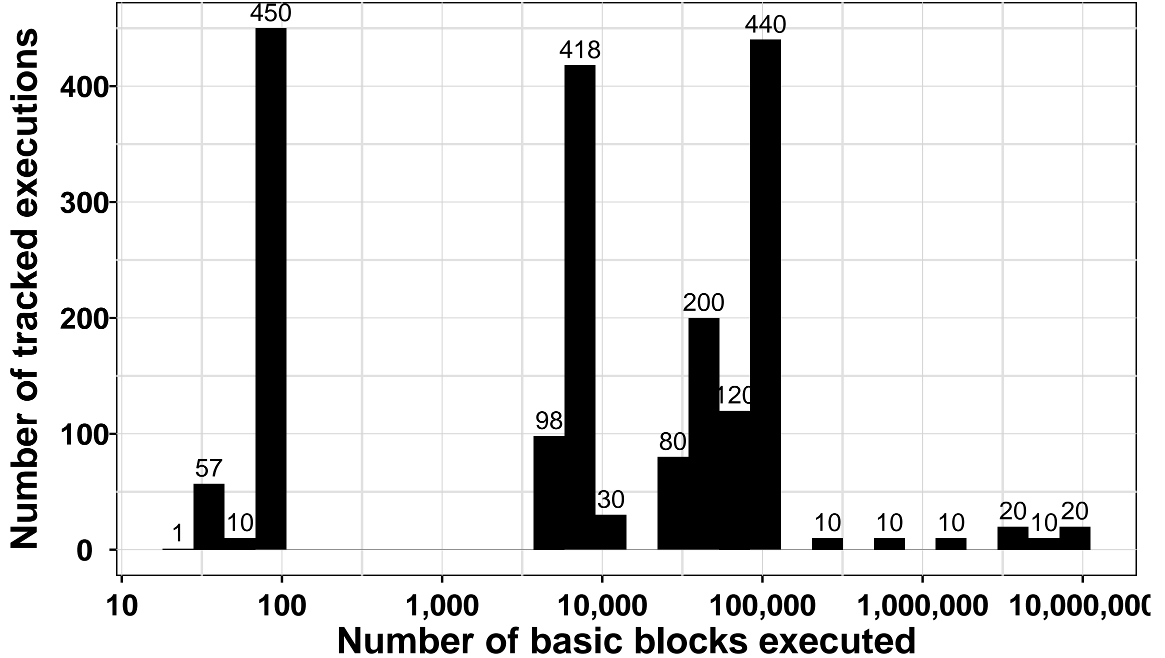

FIGURE 3.7: Total MAMBO blocks executed in Instagram.

invocations read tainted data within the first 10% of MAMBO blocks executed after a trap. The few executions near 100% are due to just a single native method.

We also present a histogram in Figure 3.7 of the total number of MAMBO blocks executed in each native code invocation. We see that about 26% execute 100 or fewer blocks and the remaining 76% execute large numbers of basic blocks. This suggests that fine-grain access control may further improve performance by delaying emulation for some invocations.

3.6.4 Energy consumption

To obtain energy measurements, we use a Monsoon mobile device power monitor. Unfortunately, the Galaxy Nexus does not allow USB pass through mode and continuously charges the battery. To overcome this, we connect the power monitor directly to the battery terminals and use WiFi-ADB with our scripting harness to invoke specific actions on the phone. For these experiments we disable auto-brightness and set screen brightness to the minimum level. We also utilize a constant wait time

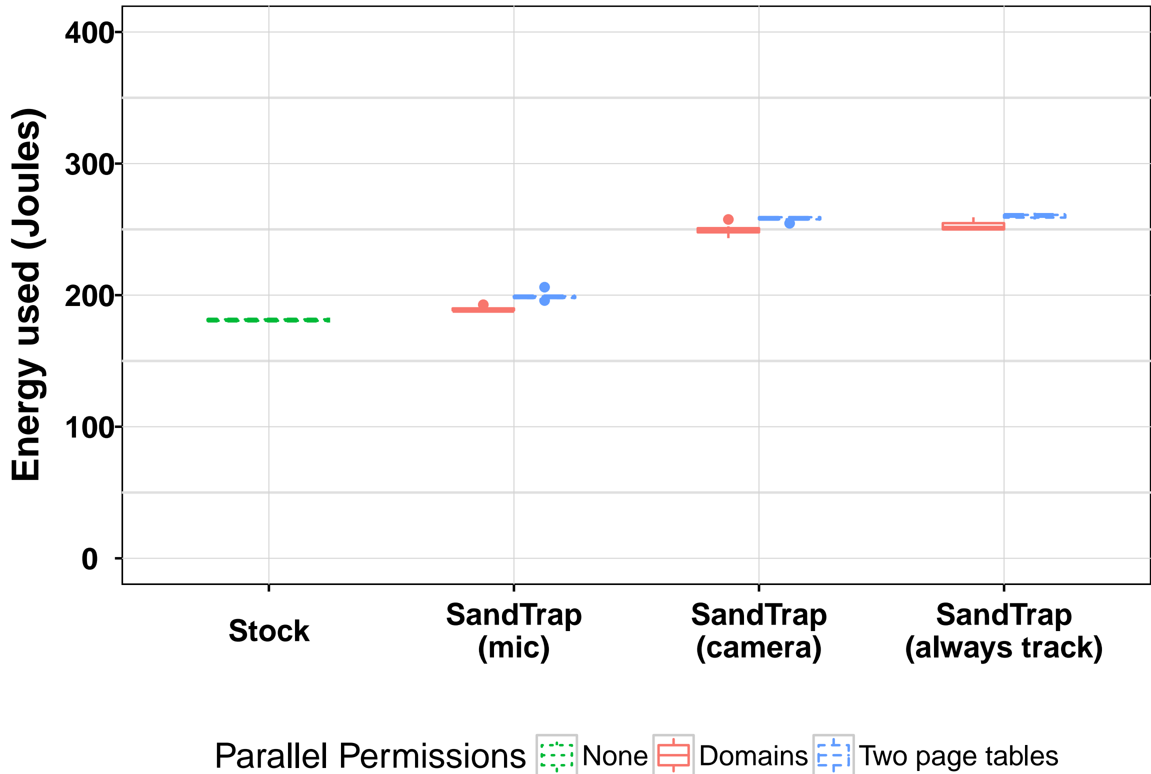

Figure 3.8: Energy consumed by the device while running the Instagram experiments under different configurations.

between screen transitions in Instagram across all experiments.

The Monsoon voltage level is set to 4V and we gather samples of current (Amps) at 5,000 samples/sec. We average the current within each one second interval and multiply it by 4V to get average power as well as energy (Joules) for the one second interval. Summing over the entire experiment duration provides the total energy consumed. Importantly, this experiment captures the workload's parallelism available by executing on multiple cores.

Figure 3.8 shows the energy results. We compare stock Android, SandTrap with the microphone source, SandTrap with the camera source, and SandTrap with DIFT for all native code. As expected, SandTrap energy increases with increased DIFT activity. Stock Android consumes a median energy of 181 Joules while SandTrap with the microphone source uses slightly around 200 Joules. Recall, Instagram does

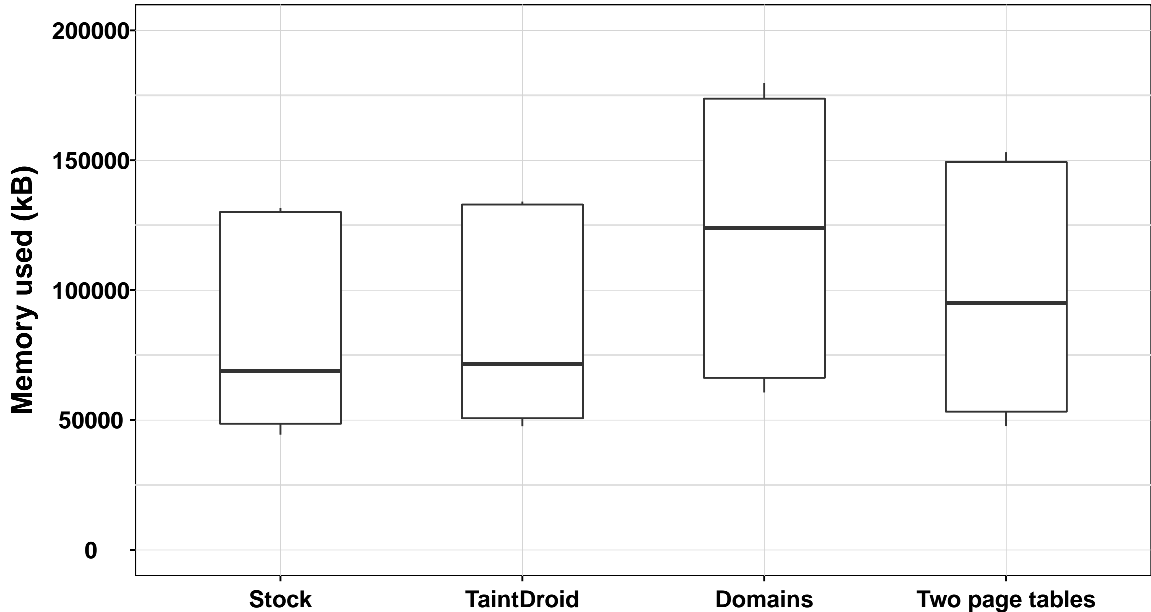

FIGURE 3.9: Memory consumed by SandTrap while running Instagram experiments under different configurations (tracking camera data).

not process any tracked data from the microphone. SandTrap with the camera source and SandTrap performing DIFT on all native code each consume around 250 Joules. Note that in all experiment types, SandTrap with the memory domains implementation of parallel permissions consistently consumed 10 Joules less than the two page tables implementation.

3.6.5 Memory domains vs two page tables

In the experiments above, memory domains outperform two page tables for several reasons. First, under domains, when a thread switches from tracked to untracked, the kernel only needs to change the DACR. Under page tables, the kernel must update the page table base register and perform additional ASID and cache-management tasks. Second, with two page tables the kernel has to synchronize table mappings. Finally, the two page tables implementation reduces TLB re-use since each table is associated with a separate ASID.

On the other hand, using memory domains is not strictly better than managing

two page tables. We ran our Instagram experiments with stock Android, TaintDroid, and SandTrap with the camera source, and collected memory use information with `procrank`, a tool on Android that shows each app's unique set size (the amount of memory private to that app). We sampled this information at each discrete point of our experiment (before taking a photo, before annotating the image, before uploading it, etc.). Therefore, our results represent the steady state memory usage. The domains implementation of SandTrap consumes the most memory, using a median of 120 MB of memory, compared to 90 MB for page tables SandTrap, 70 MB for TaintDroid, and 68 MB for stock Android. Recall that when a 1 MB region of memory is placed in the `TRACKED` domain, SandTrap immediately allocates a new copy of all the pages that are shared or mapped to the zero page.

3.7 Related work

Section 3.2 describes SandTrap's relationship to prior work on DIFT. In this section, we focus on SandTrap's relationship to prior work on memory protection and sharing. Mondrian [66], CHERI [67], and proposals for unlimited watchpoints [38] utilize new or specialized hardware to provide per-core memory permissions. These systems would be useful for implementing on-demand DIFT, but we designed SandTrap with existing commodity ARM processors in mind.

Wedge sthreads [12] and lightweight contexts (lwCs) [46] allow developers to refactor their applications into protected components. Both sthreads and lwCs manage memory protections by creating a new page table for each execution context. This is similar to SandTrap's two page table approach.

ARMLock [70] and Shreds [18] use memory domains' no-access mode to protect regions of memory from components that share an address space. In contrast, SandTrap uses manager mode to allow threads to bypass page protections and access protected pages. Recent work on shared address translation for Android [30] used

memory domains to improve fork performance by consolidating the use of physical pages for storage page tables.

Finally, Dune [11] uses x86 virtualization hardware to give user-level code control over its own page table, but Dune's page table management is not thread safe, and ARM is listed as future work.

3.8 Conclusion

In this work we have presented the design and implementation of SandTrap. SandTrap performs on-demand DIFT of third-party native libraries in Android using parallel permissions. Experiments with a prototype implementation demonstrate that tracking overhead for native code is proportional to the amount of tainted data it handles.

4

DoubleVision: Protecting Application Resources with Parallel Permissions

4.1 Introduction

Stray memory accesses are at the root of numerous application bugs. These are caused by common coding mistakes [21–27], such as an incorrect array boundary test that leads to a read or a write past the end of an array. If these stray memory accesses reach an unmapped portion of the process's address space (AS), the process would simply segfault. However, if they reach memory-mapped application resources, the integrity or confidentiality of those resources may be violated.

For example, in an application that loads an SQLite database as a memory-mapped file, stray writes into virtual addresses used by the database will corrupt it. Stray reads are also harmful. If an application uses RSA private keys, a stray read may inadvertently leak the keys. An infamous instance of this latter issue is the HeartBleed vulnerability in OpenSSL; an incorrect bounds checking in its implementation of the TLS heartbeat protocol can cause a stray read into the private keys loaded by the process, which will then be copied and sent out to clients.

Prior work in protecting sensitive data in processes propose new OS primitives, either to separate applications into different components [12, 46], or to isolate sensitive and non-sensitive objects in memory [18, 65]. In the former approach, each component is assigned a different set of privileges, which are then enforced by the OS. In the latter, threads only have access to sensitive data when they are executing trusted code. Given that these prior work were developed with a focus on adversarial scenarios, the protections they provide are comprehensive. However, they come at a cost of non-trivial refactoring of existing application code [12, 46], or needing custom compiler toolchains and special hardware features [18, 65].

We take a different approach. Instead of trying to solve the problem in the general case with sophisticated adversaries, we focus on the regular, if banal, case of developers inadvertently corrupting or leaking their own resources due to bugs in their code. We sought to answer the following question: how can the resources of commonly used libraries, such as SQLite and OpenSSL, be protected from stray memory accesses without requiring special hardware features or significant code refactoring?

In this paper, we present *DoubleVision*, a system that attempts to answer this question using *parallel memory permissions*. Using DoubleVision is straightforward. First, developers need to designate the portions of the address space occupied by the application resource as protected. Next, they need to call `acquire()` just before the resource is accessed. Finally, immediately after the read or write to the resource is completed, they need to call `waive()`. The calls to `acquire()` and `waive()` grant and remove the calling thread's permissions to the protected resource, respectively.

Consider a modified version of the SQLite library that uses DoubleVision to protect memory-mapped databases from stray writes. First, we observe that an application using this modified library is unlikely to access the database via raw pointers. Instead, it will use methods in the SQLite API such as `blob_read()`. Therefore, the application will not contain calls to `acquire()` and its stray writes

to the portion of the AS occupied by the database will be caught. Second, we observe that even within the SQLite library, it is easy to identify which functions need write access; only those that write into the database, such as `blob_write()`, should call `acquire()`. Therefore, stray writes from other portions of the library will be caught as well. DoubleVision will not be able to prevent stray accesses within an `acquire()` / `waive()` pair. We consider such situations rare given that the principle of least-privilege applies here; developers should only place `acquire()` / `waive()` pairs around code that need access to the protected resource.

We implemented DoubleVision on a Raspberry Pi 3 device by modifying the Linux kernel to provide parallel memory permissions. We evaluated our prototype by running application benchmarks on top of modified versions of libraries such as SQLite, OpenSSL, and LMDB. Modifying these libraries involved including a C header file to the project, and changing the salient portions of the libraries to set protections and call `acquire()` / `waive()`. The C header file is less than 100 lines and contains DoubleVision wrappers the libraries can call. The library modifications are in the order of tens of lines of code. No modifications were required to the applications using these libraries. In our experience, the main challenge in retrofitting these libraries to use DoubleVision was in understanding their respective code bases. We expect those with more familiarity with the internals of these libraries will take less time than us to perform similar modifications.

We consider the following to be our key contribution with DoubleVision: we show that it is possible for applications to protect memory-mapped resources from buggy stray memory accesses without requiring significant application refactoring or special hardware features.

4.2 Background

4.2.1 Memory-Mapped I/O

An application can access files stored on disk either using the `read()` / `write()` syscalls, or by memory-mapping the file into its AS (via `mmap()`) and then directly reading and writing to memory. We refer to these I/O methods as `rwio` and `mmio`, respectively. There are a number of reasons why `mmio` generally provides better performance over `rwio`: (1) the total number of syscalls used is reduced, since there is no need to repeatedly call `read()`, `write()`, or `lseek()` (used to reposition file descriptors), (2) `mmio` reduces the number of buffer copies between user space and kernel space, and (3) `mmio` avoids certain portions of the kernel's filesystem stack when persisting the file to disk.

We ran a series of experiments with IOZone [3], a file system benchmarking tool, to measure the performance gains of `mmio` over `rwio` in a running system. Since DoubleVision is currently implemented for the Raspberry Pi 3 (henceforth referred to simply as "Pi"), for consistency's sake, we chose to run these experiments on the Pi as well. In these experiments, we used Raspbian [7], the default Linux distribution for the Pi, with Linux kernel v4.9.61, the `ext4` filesystem, and a 64 GB microSD card for storage.

We investigated the performance of writing and reading from a 16 MB file in varying block sizes. In the `rwio` configuration, we configured IOZone to open the file with the `O_SYNC` flag so that calls to `write()` do not return until the changes are persisted to disk. Similarly, in the `mmio` configuration, we called `msync()` with the `MS_SYNC` flag after writing to the memory locations mapped to the file. We ran the experiment ten times, and plot the speedup of `mmio` over the median of `rwio`.

In Figure 4.1, we show the speedup of using `mmio` in read-only benchmarks (described in Table 4.3). In general, `mmio` is faster, particularly when block sizes are less

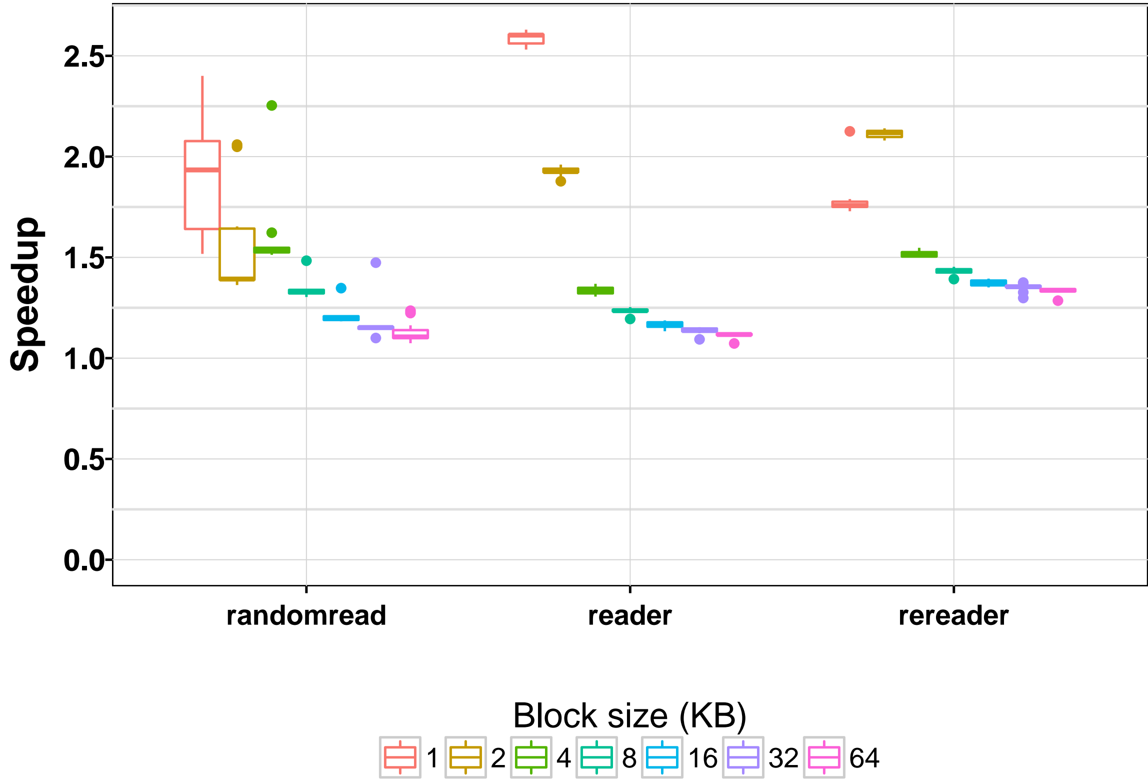

FIGURE 4.1: Speedup of using `mmio` over `rwio` when reading from a file in IOZone benchmarks.

than the page size (4 KB). The Linux kernel typically defers the task of loading a file into memory when an application requests it. Instead, it waits for the traps caused by the application's accesses to unloaded portions of the file, using that opportunity to load just the required file regions into physical memory pages. Subsequent accesses to the parts of file already loaded do not trap. This explains why `mmio` read performance is substantially better when block sizes are less than the page size.

Figure 4.2 illustrates the speedup of using `mmio` in write-only benchmarks. The speedups are benchmark-dependent. When writing sequentially to a new file, the performance of `mmio` reaches `rwio` as block sizes increase. However, when writing sequentially to an existing file, the performance of `mmio` is several times better than `rwio` regardless of block sizes. We investigated this closely by running a separate set of experiments together with `ftrace`, a tool to trace kernel activity. We were

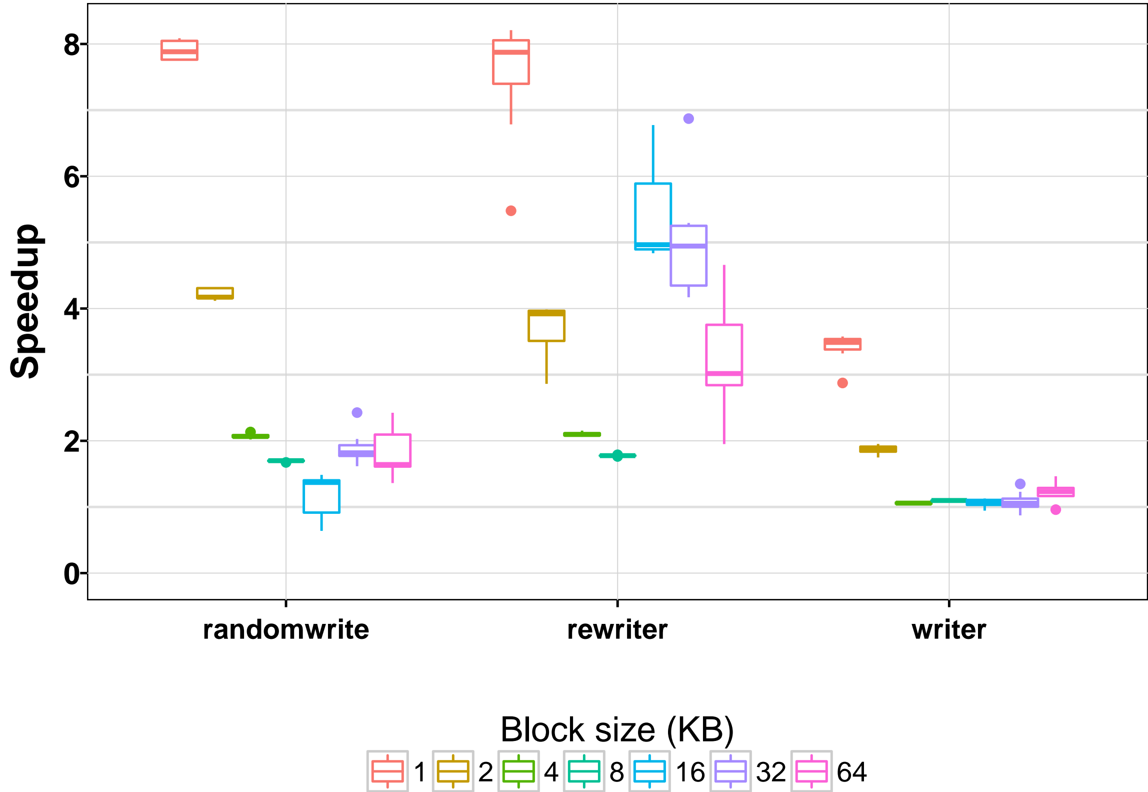

FIGURE 4.2: Speedup of using `mmio` over `rwio` when writing to a file.

particularly interested in the behavior of the `msync()` and `write()` syscalls.

We found two plausible reasons for the increased performance: (i) when writing to an existing file, the kernel does not need to allocate additional metadata to keep track of the file in the file system, and (ii) when persisting the writes to disk, `mmio` and `rwio` take different code paths through the file system stack. For example, the `msync()` in `mmio` avoids `rwio`-specific operations such as copying the user buffer specified in the `write()` system call.

In summary, there are conceptual reasons why `mmio` should perform better than `rwio`, and we have shown this to be true in practice using several IOZone experiments. Recent work has also shown that `mmio` has room for improvement [19], by providing memory-access hints to the kernel (via `madvise()`). That said, note that file system benchmarking, which we have done here, is a complex affair [64]. It depends on

a variety of factors, including application workload, the file system used, and the underlying storage layer's I/O speed. In some cases, bottlenecks in other parts of the system can negate the benefits of `mmio`. For example, we observed that in write benchmarks with 64 MB block sizes, the Pi's sdcard I/O speed became the bottleneck. Therefore, we do not claim that `mmio` *always* performs better than `rwio`.

4.2.2 Stray memory accesses

A potential downside of using `mmio` is the problem of stray memory accesses, where application bugs can cause accidental reads or writes to parts of the AS that are memory-mapped to files or other resources. Developers of popular libraries are aware of this problem and have reacted accordingly. The SQLite library "uses a read-only memory map to prevent stray pointers in the application from overwriting and corrupting the database file" [60]; the BoltDB library considers it "unsafe" to map the database with write permissions [13] and disallows it, because "[mapping the database with write permissions] is really dangerous if you have any bugs in your code. You're essentially writing directly to the underlying data file" [14]; the LMDB (Lightning Memory-mapped Database Manager) library states that its "[in-memory database] can be used as a read-only or read-write map. It is read-only by default as this provides total immunity to corruption" [47].

These developers are aware that using `mmio` with write permissions offer better performance, but choose to either disable it by default or disallow it completely for the sake of safety. As the introduction to LMDB states: "Using read-write mode offers much higher write performance, but adds the possibility for stray application writes thru (sic) pointers to silently corrupt the database. Of course if your application code is known to be bug-free (...) then this is not an issue" [47].

We investigated LMDB's performance speedup when the database is mapped with write permissions using several write-only micro-benchmarks provided by the

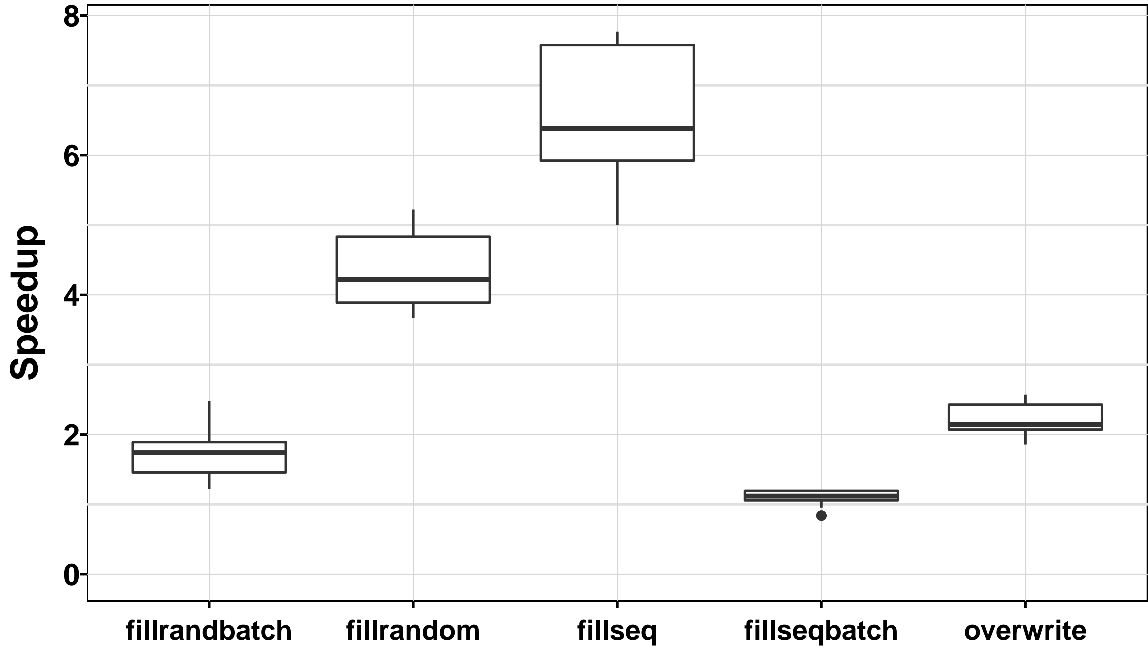

FIGURE 4.3: Speedup of using mapping databases with read-write permissions instead of just read in LMDB.

developers [40, 62]. Our results are illustrated in Figure 4.3. Once again, we observe that having write permissions to the memory-mapped database offers better performance. Note however that some of the speedup here is attributable to the fact that the writes are performed asynchronously. There is no immediate flush to disk via an `msync()` after each write.

In the synchronous write-only LMDB micro-benchmarks (not shown in Figure 4.3), the performance of `mmio` and `rwio` were close. This conflicts with our finding in Section 4.2.1, that there are performance gains to be had with `mmio` even when synchronous writes are used. This suggests to us that the internal design of the LMDB library does not take full advantage of `mmio` writes, perhaps because there was no way to do so without sacrificing safety. We believe that if developers had access to a memory protection scheme that provides the benefits of `mmio` writes without the associated safety issues, they would update the design of their libraries accordingly.

While we have focused on stray writes in this section, stray reads are also harmful,

albeit less catastrophic at first glance. The HeartBleed vulnerability in OpenSSL is an example of how stray reads within a library can leak sensitive data. Even in applications that only use one of the aforementioned database libraries, it may be helpful to catch stray reads to the database. After all, such accesses are application bugs and may be symptomatic of other issues. Furthermore, the database itself may contain sensitive data. In summary, we regard all stray memory accesses to the portions of an AS mapped to application resources as undesirable.

4.2.3 Page protections

As explained earlier in Chapter 2, page protections apply to every thread in a process. We may not use page protections alone to protect resources in multi-threaded applications unless we limit parallelism and prevent the threads that require access to the protected pages and the threads that do not from executing in parallel. As in SandTrap (Section 3.2.2), we consider this unacceptable in DoubleVision. Modern CPUs are predominantly multi-core and applications have grown to take advantage of them.

4.3 Design

In this section, we discuss the design principles that guided DoubleVision.

4.3.1 Focus on buggy stray accesses

In DoubleVision, we focus on the problem of stray memory accesses due to bugs. As quoted in Section 4.2.2, existing database libraries such as SQLite and BoltDB have either disabled memory-mapped databases with write permissions by default or disallowed it altogether due to potential corruption. We believe there is a need for a solution that obviates the need for such workarounds so that libraries can gain the performance of `mmio` without sacrificing safety. We also believe that focusing on stray

memory accesses due to bugs will yield a simple solution. Therefore, in DoubleVision, we do not address the problem of malware and other adversarial threats.

4.3.2 Amenable to existing applications

We wanted it to be easy for developers to reason about the protections offered by DoubleVision and for them to modify their existing applications and libraries accordingly. This meant to us that we could not significantly alter well-understood concepts such as processes and threads. To illustrate this principle better, we present Wedge [12] and light-weight contexts (lwCs) [46].

Wedge provides the ability to compartmentalize applications using tagged memory and *sthreads*. When applications allocate objects in memory, they may associate them with different tags. Next, they can start an sthread for execution, and specify which tags it may access. The sthread creator may also specify which file descriptors are to be shared with the sthread. Aside from these explicitly shared resources, sthreads and their parent processes have their own copy of program state.

As discussed earlier in Chapter 2, lwCs provide an abstraction that allows an application to be broken into separate components, each with its own set of permissions. Just like processes, each lwC has its own virtual memory mappings and file descriptors. However, unlike processes, they are not schedulable entities; instead a thread chooses an lwC to execute under and may later switch to a different lwC. Unless explicitly requested, lwCs do not share memory mappings with other lwCs. As the authors demonstrate, this abstraction can be used for a variety of purposes, including the protection of private keys in OpenSSL.

The primitives provided by Wedge and lwC differ in a key way compared to regular threads: sthreads and lwCs have to explicitly share resources with each other. Otherwise, they have separate address spaces. We believe that this may restrict adoption as developers need to carefully reason about these primitives and make

non-trivial modifications. For instance, before they break their existing applications into separate components, they have reason about all program state and decide what needs to be shared. In multi-threaded apps, the presence of synchronization primitives can make this tricky, as it is important to not introduce deadlocks. Wedge provides tools to help with some of these modifications but their usability among developers of varying expertise levels is not well-understood.

Unlike Wedge and lwCs, we opted for *default-share* semantics. That is, all threads share the same resources by default. Only the portions of the AS explicitly marked would be protected from stray accesses. We found that this helped significantly in quickly porting existing libraries to use DoubleVision, because we only needed to reason about program state that needs to be protected. Note that the choice of *default-share* is possible only because of our focus on buggy stray accesses (as opposed to malicious accesses).

4.3.3 Compatibility with existing hardware

We wanted to avoid solutions that required special hardware features beyond what most platforms already provide. We do not make any assumptions on where parallel permissions may be used. Some developers may opt to use it in server-class machines while others may find it more useful in IoT devices. Therefore, it was extremely important to us that our techniques were general. In contrast, consider Shreds [18] and ERIM [65].

In Shreds, applications may place sensitive data in memory into different *s-pools*. When a thread needs to access protected data, it will enter a *shreds* mode, which allows it to access the *s-pool* associated with that shred. Once it is done accessing the protected data, the thread will exit the shred. Shreds requires a custom compiler toolchain, which is used to verify the safety properties of the program that uses the shreds API, and to add the instrumentation that will protect the s-pools during

runtime.

ERIM uses several techniques to isolate sensitive memory within processes. First, applications are split into trusted and untrusted components. Second, ERIM uses Intel's Memory Protection Keys (MPK) to set protections on sensitive data. Finally, ERIM allows only trusted code to access the sensitive data. It does so with a combination of static and dynamic binary rewriting, and with syscall mediations via a runtime library. This prevents untrusted code from gaining access to sensitive data, either by simply calling MPK instructions to change permissions, or by creative mapping and unmapping of memory in the process's AS.

Both ERIM and Shreds are close to DoubleVision in spirit, given their focus on protecting memory without modifying processes or threads. However, they both rely on custom hardware features beyond page protections. The current prototype of Shreds uses ARM's memory domains feature, which is only available in ARM CPUs running in 32-bit mode. The authors claim it may also be built on top of Intel MPK, which ERIM uses. As we noted in Chapter 2, the issue with Intel MPK is that it is only available for select Intel x86 CPUs at the moment, and it is not clear whether AMD's x86 processors or ARM CPUs running in 64-bit mode will have similar features. In any case, page protections alone, which are available in all modern CPUs, do not suffice in either work.

4.3.4 Must not limit parallelism

Finally, we reject any solution that would prevent threads in a process from executing in parallel. Even IoT-class devices, such as the Raspberry Pi, have multiple CPU cores. If we limit parallelism, we would severely impact performance in multi-threaded applications. As described in Section 4.2, we are motivated in enabling developers to use `mmio` to gain its associated performance benefits without sacrificing safety. If DoubleVision negates the performance benefits of `mmio` over `rwio`, then

there would be no reason for developers to use it.

4.4 DoubleVision

In this section, we present an overview of DoubleVision and its implementation.

4.4.1 Parallel memory permissions

The key building block in DoubleVision is *parallel memory permissions*. DoubleVision uses the two page table implementation since one of our design goals is to have a general solution that may work on a variety of hardware platforms. Observe that our *default-share* semantics meshes well with what parallel memory permissions already provide, since it synchronizes the virtual-to-physical mappings in the two page tables.

4.4.2 Types of protections

How applications or libraries use DoubleVision depends on their requirements. If they only require protection from stray writes, then they merely need to set read-only permissions on the virtual address occupied by the resource. They will do so either on page table tbl_A or tbl_B, and switch page tables accordingly when write permissions are required. Similarly, if they need protection from both stray reads and writes, they need to remove both read and write permissions in one of the two tables, and switch tables when they need to access the resource. When a stray memory access occurs, the kernel will send a signal to the thread that made the erroneous access. If the signal is unhandled, the process will be terminated as though it segfaulted.

Observe that since DoubleVision uses two page tables per process, it can also modify the virtual-to-physical memory mappings for certain virtual address ranges. That is, instead of setting page permissions, it can allocate new scratch physical pages in memory, zero them, and map those to the virtual address ranges occupied

by the resource. In this scenario, stray reads and writes will not trap but they will not affect the protected resource either; instead they will proceed to scratch memory.

This is akin to failure oblivious computing [56], where a failure in an application (the stray accesses) does not immediately crash the process. While this does not address the underlying bug that caused the stray access, it defers the crash of the process. This may be suitable for long-running processes such as server-side applications. We leave it to the developer to decide if this is suitable for their workload. If they prefer fail-stop failures, they can rely on page protections.

We designed DoubleVision to handle stray user-space accesses. However, it is also possible for stray memory accesses to occur within kernel space, such as by buggy device drivers or kernel modules. Our current prototype supports setting page permissions to restrict kernel accesses to virtual pages. However, we have not tested it thoroughly and leave it to future work.

4.4.3 API

Table 4.1: DoubleVision API

Syscall	**Description**
`mprotect_b(addr, len, prot)`	In page table tbl_B, set the protections specified in `prot` on the virtual address range [`addr`, `addr+len`).
`mmap_b(addr, len)`	In page table tbl_B, allocate new memory and map them to the virtual address range [`addr`, `addr+len`).
`munmap_b(addr, len)`	In page table tbl_B, unmap memory allocated by DoubleVision and use the same mappings as in page table tbl_A for the virtual address range [`addr`, `addr+len`).
`switch_table(tbl)`	Switch the page table used by the calling thread.

Parallel memory permissions in DoubleVision is exposed to the application as new syscalls, which are summarized in Table 4.1. Note that although this API is more expressive than the API in Table 2.1, it is still fundamentally parallel memory

permissions. Some of its expressiveness is a result of modeling it closely after the existing `mprotect()`, `mmap()`, and `munmap()` syscalls (which modify page table tbl_A), and because we use the two page tables implementation. To make it easy for developers to use DoubleVision, we also provide a single C header file which has wrappers for these new syscalls.

The exception to DoubleVision's *default-share* semantics is when an application uses `mmap_b()` to establish new memory mappings in page table tbl_B for a given virtual address range. DoubleVision will not synchronize virtual-to-physical mappings in this range until the application calls `munmap_b()`.

If a process calls `fork()` and creates a new process, the new process inherits all parallel memory permissions and mappings from the parent. By default, all new processes and threads start out using page table tbl_A. However, they will use tbl_B if a thread created them while using tbl_B.

The following describes how a developer might use our API:

1. After loading a resource into memory, she will set the required protections on the resource's virtual address range with `mprotect_b()` or `mmap_b()`. Next, she will switch to page table B with `switch_table()` to immediately prevent stray accesses.

2. The `acquire()` and `waive()` calls described in Section 4.1 are implemented using the `switch_table()` syscall. On an `acquire()`, the thread will switch to tbl_A and on `waive()`, it will revert back to page table tbl_B. The developer will place `acquire()` / `waive()` pairs in her code so that only the appropriate regions have permissions to access the resource.

3. If the developer needs to, she can modify or remove the protections set in page table tbl_B with calls to `mprotect_b()` or `munmap_b()`.

4.4.4 Limitations

Although DoubleVision has a straightforward implementation and an API that is amenable to existing applications, it is not without its limitations.

Coarse granularity

Since DoubleVision relies on page permissions, there may be a need to modify memory allocation code so that the protected data does not share a page with other data that must be freely accessible. For example, when we modified OpenSSL to use DoubleVision, we changed its private key allocation routines to use `mmap()` instead of `malloc()`. This ensured that by setting page protections, we did not inadvertently also prevent the library from accessing other application state allocated in the heap. However, modifying the library to use `mmap()` instead of `malloc()` may not always be feasible.

Consider the case of the HeartBleed; because of stray reads, the vulnerable code would copy random program state from the webserver's AS and send it to the client. This state includes both the RSA private keys loaded by OpenSSL, and other application-specific data such as username, passwords, and private messages users send to each other. By modifying OpenSSL, DoubleVision can protect the RSA private keys from leaving the system. However, it cannot prevent the application-specific data from leaving the system unless the web page backend runtime also allocates these sensitive data in different memory pages and sets the appropriate protections.

One way of mitigating this limitation is to use custom heap allocators that are aware of parallel memory permissions and place objects with the same permission requirements in the same physical page. It will be straightforward to replace calls to `malloc()` to this custom heap allocator. However, this does not address the limitation completely. First, a thread with permissions to access a small object in a

page can perform stray reads or writes to other objects in the same page. Second, it is simply unable to handle the case when an application needs to place different permissions to contiguously allocated memory in word-size granularities. We leave the development of these custom heap allocators to future work, and leave it to developers to decide if DoubleVision is appropriate for them.

Dual protections

As a consequence of using two page tables per process, the current prototype of DoubleVision offers limited protections: one page table has to allow full access while the other disallows it. This may not be adequate in complex applications that use multiple libraries. For example, an application that uses both SQLite and OpenSSL may set protections in page table tbl_B on the portions of the AS used by the in-memory database and RSA private keys, and use page table tbl_B by default. If a thread switches to page table tbl_A while executing SQLite code to access the database, bugs in the SQLite library can cause stray accesses to the RSA private keys and these would not be caught. We need three page tables in this example to properly isolate OpenSSL and SQLite resources. In general, to independently protect n resources, we need $n + 1$ page tables.

We believe implementing more than two page tables would not be difficult. However, having more page tables will increase the amount of overhead since the kernel has to allocate additional space for the tables and synchronize them all. We leave it to future work to implement more than two page tables and provide the appropriate API so that they may be used easily.

4.5 Evaluation

In this section, we present our evaluation of DoubleVision. We sought answers for the following questions: 1) How difficult is it to modify existing libraries and applications

to use DoubleVision? 2) What are the performance benefits of using `mmio` over `rwio` while using DoubleVision? 3) How much overhead does DoubleVision impose?

We attempted to answer these questions by benchmarking modified versions of OpenSSL, SQLite, LMDB, and IOZone. We implemented and evaluated DoubleVision on a Raspberry Pi 3, which has a quad-core 1.2 GHz CPU and 1 GB of RAM, while running Raspbian, a 32-bit Linux distribution based on Debian, with Linux kernel v4.9.61. Note that in the following experiments, we often compare with a stock baseline system. This refers to a configuration where we used an unmodified version of the respective library and the kernel.

4.5.1 Modifying libraries

Table 4.2: Summary of our modifications to libraries to protect resources with DoubleVision.

Library	**Version**	**Change summary**
OpenSSL	1.0.1e	7 files changed, 119 insertions(+), 17 deletions(-)
LMDB	0.9.21 (git commit 4d5e2d2)	1 file changed, 39 insertions(+), 10 deletions(-)
SQLite	3.21.0	1 file changed, 32 insertions(+), 1 deletion(-)
IOZone	3.471	1 file changed, 7 insertions(+)

In Table 4.2, we summarize our modifications to the libraries we used. We excluded DoubleVision's 85 line C header file from the summary. In general, our modifications were minimal as we only had to set protections with `mprotect_b()` and switch page tables appropriately with `switch_tbl()`. OpenSSL required more changes for two reasons: 1) we had to change the memory allocation of the RSA private keys to use `mmap()` instead of `malloc()` (around 37 lines in `crypto/bn/bn_lib.c`), and 2) we carefully added calls to `switch_tbl()` at the right places, so that the RSA private keys are accessible only when absolutely necessary.

We verified that our changes were correct by means of manual testing. Since we were able to successfully modify these libraries without having prior knowledge

of their internals, we believe developers will also find it easy to use DoubleVision. Note that we used the latest available version of each library. The exception to this was OpenSSL, where we deliberately chose a version vulnerable to the HeartBleed attack.

4.5.2 DoubleVision and `mmio`

Table 4.3: Description of IOZone & LMDB Benchmarks.

Application	**Name**	**Description**
IOZone	writer	Sequentially write to a new file.
	rewriter	Sequentially write to an existing file.
	randomwrite	Randomly write to an existing file.
	reader	Sequentially read from an existing file.
	rereader	Sequentially read from a recently read file.
	randomread	Randomly read from an existing file.
LMDB	fillrandbatch	Batch write values asynchronously in random key order.
	fillseqbatch	Batch write values asynchronously in sequential key order.
	fillrandom	Write values asynchronously in random key order.
	fillseq	Write values asynchronously in sequential key order.
	overwrite	Overwrite values asynchronously in random key order.

We re-ran the IOZone experiments described earlier in Section 4.2.1 using DoubleVision to protect against stray writes (the benchmarks are described in Table 4.3). In Figure 4.4, we show the speedup of using `mmio` (w/ DoubleVision) over stock `rwio` in write-only benchmarks. Even with protections enabled, `mmio` offers significant speedups over stock `rwio`.

In Figure 4.5 and in Figure 4.6, we compare DoubleVision `mmio` against the median of stock `mmio` in write and read benchmarks, respectively. Intuitively, we expect to see some amount of overhead in the write benchmarks since IOZone has to call `acquire()` and `waive()` (both implemented via `switch_tbl()`), before and after writing files, and we expect to see minimal overhead in the read benchmarks since we are not protecting against stray reads. However, our results show minimal

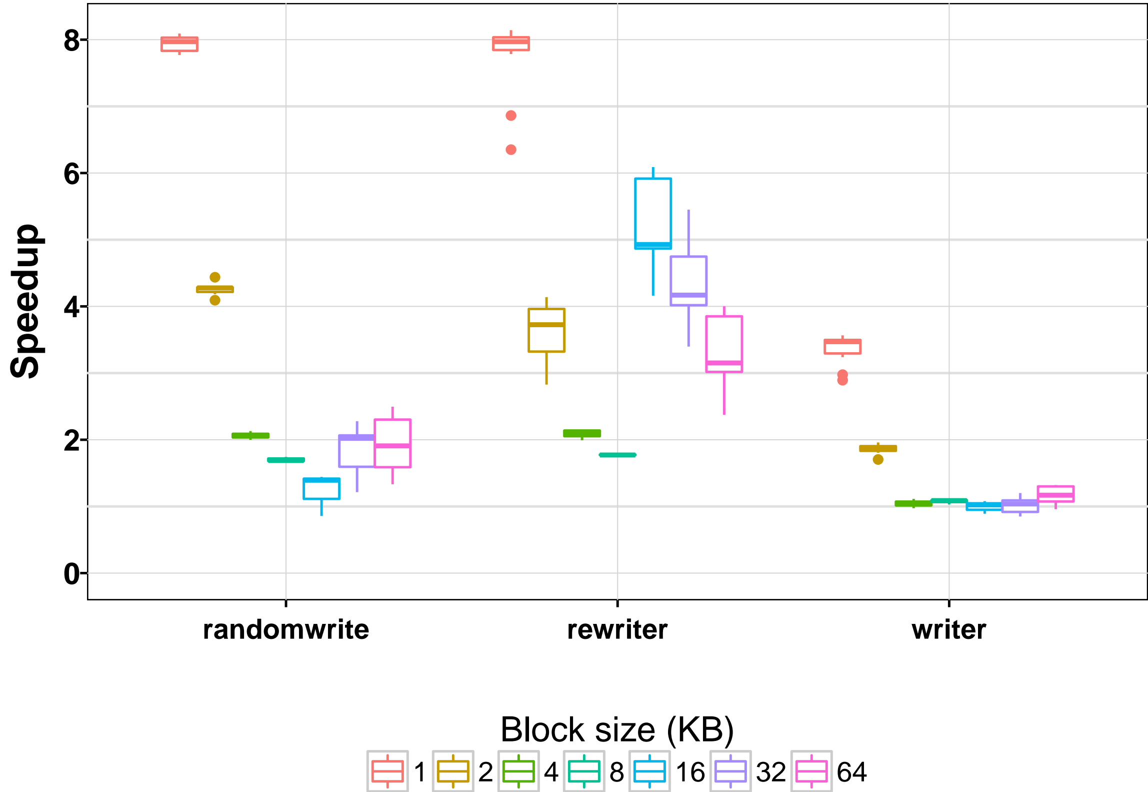

FIGURE 4.4: Speedup of using `mmio` (w/ DoubleVision) over stock `rwio`.

overhead in both cases. In fact, in some cases, DoubleVision `mmio` appears to be faster than stock `mmio`.

This may be due to I/O. Recall from Section 4.2.1 that in the write benchmarks, we call `msync()` to persist the writes, and that in the read benchmarks, the kernel loads different regions of a file from disk only when they are first accessed. We believe that the work required to perform this I/O eclipses the overhead imposed by DoubleVision, and that the variation in I/O speeds accounts for why DoubleVision `mmio` is sometimes faster than stock `mmio`.

4.5.3 DoubleVision overhead

In this section, we present our experiments with LMDB, SQLite, and OpenSSL, to better gauge the impact of DoubleVision in commonly-used libraries.

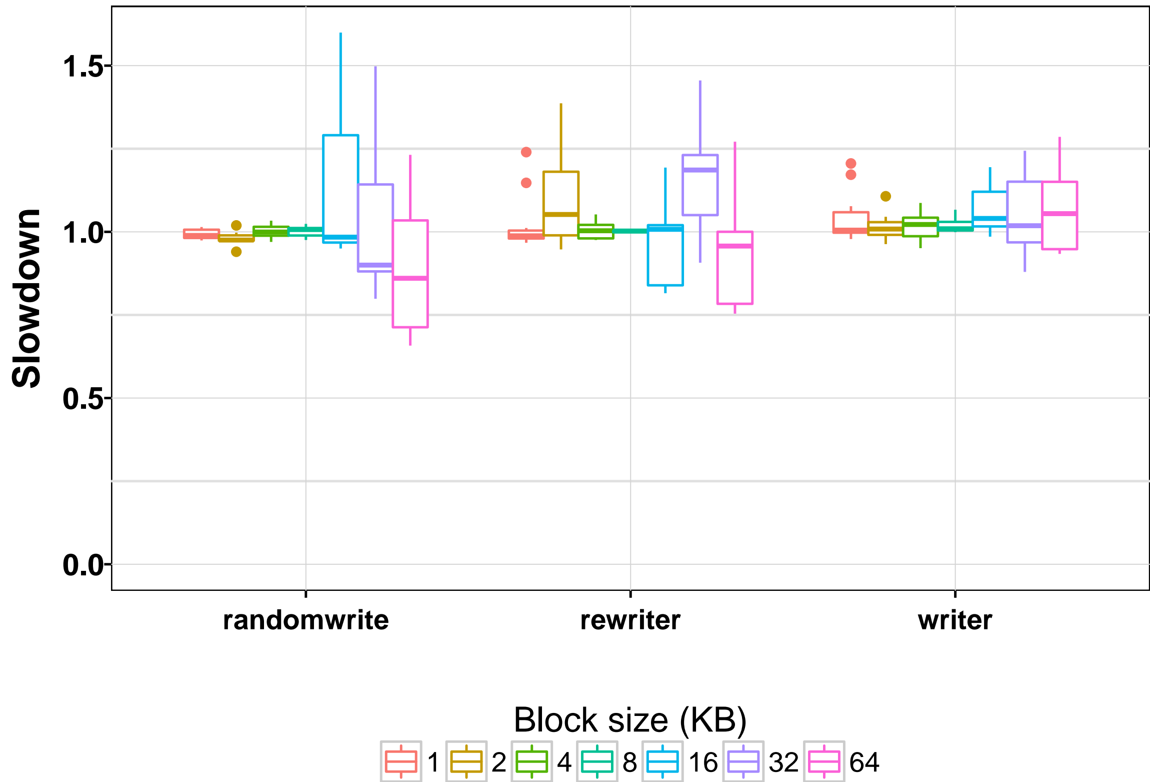

FIGURE 4.5: Slowdown of using DoubleVision `mmio` over stock `mmio` when writing to a file.

LMDB

LMDB is an in-memory key-value store. We modified it to use DoubleVision to protect against stray writes, and tested it with a series of benchmarks (see Table 4.3). These benchmarks were originally developed for the LevelDB [4] library but were adapted to LMDB by the developers so that the different database libraries can be compared with each other [40, 62].

We ran these benchmarks using a database with 1,000,000 entries, 16-byte keys, and 100 byte values. The total size of the database is about 116 MB. By default, LMDB maps the database into the AS with the pages marked with read-only permissions. Read-write permissions may be enabled manually during database load time. In Figure 4.7, we compare our modified LMDB (with a write-mapped database and

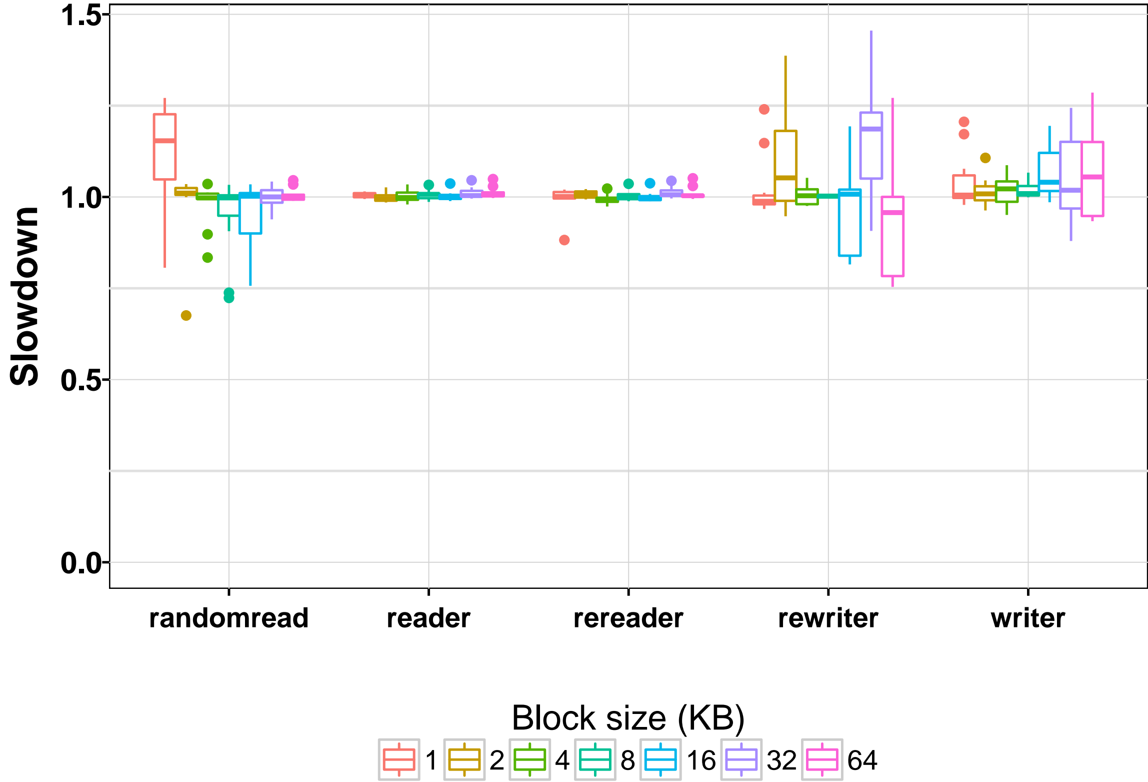

FIGURE 4.6: Slowdown of using DoubleVision `mmio` over stock `mmio` when reading from a file.

the DoubleVision kernel) against stock LMDB (database mapped without write permissions and the stock kernel). Our results show that DoubleVision can indeed enable write-mapped databases without sacrificing safety or performance. However, there are cases, such as the *fillseqbatch* benchmark, where DoubleVision performs worse than stock.

We investigated this closely by examining the raw numbers, and by comparing, in Figure 4.8, the performance of write-mapped databases with and without DoubleVision. In Figure 4.3, the performance of *fillseqbatch* in write-mapped databases without DoubleVision show only a marginal improvement over read-only databases. The overhead imposed by DoubleVision is larger than this marginal improvement, which is why *fillseqbatch* performs worse than stock in Figure 4.7. In the *fillseq*, *fillrandom*, and *fillrandbatch* benchmarks, DoubleVision imposes some overhead but

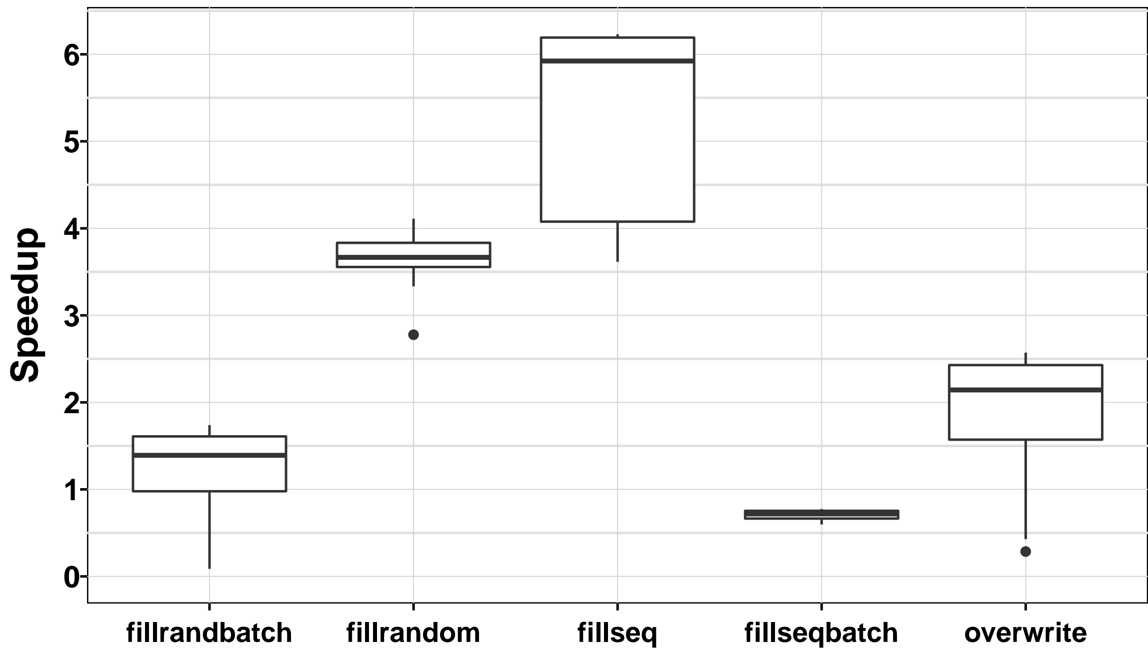

FIGURE 4.7: Speedup of using LMDB with DoubleVision enabled and the database mapped with write permissions over stock LMDB (database mapped with read-only permissions).

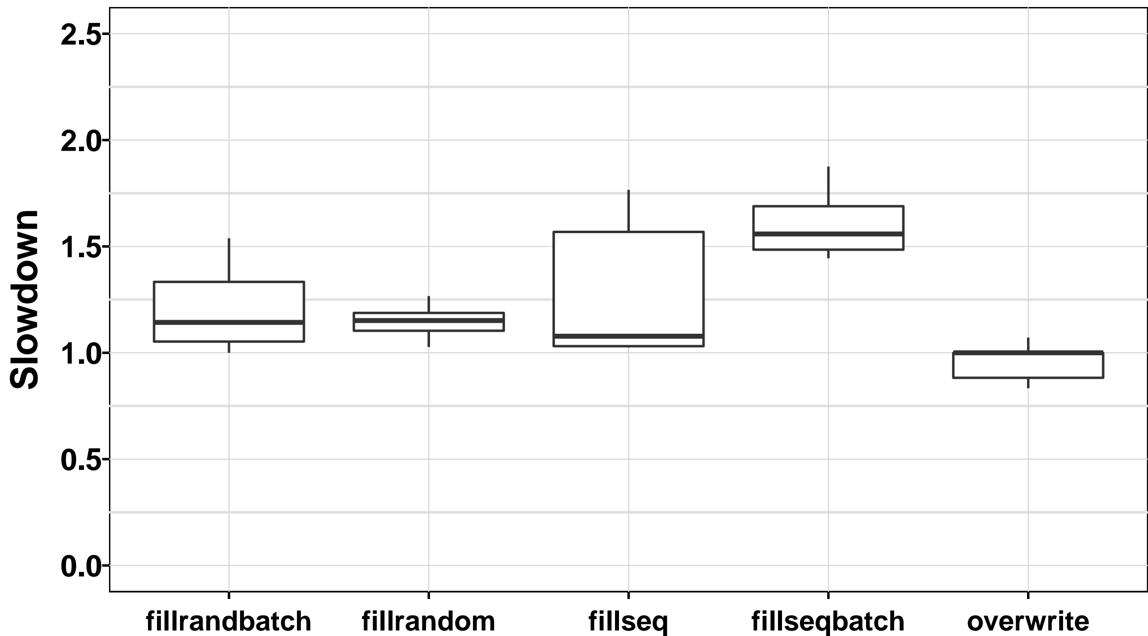

FIGURE 4.8: Slowdown of DoubleVision LMDB over stock LMDB when the database is mapped with write permissions in both cases.

they do not negate the performance gained by using write-mapped databases. Finally, in the case of *overwrite*, we believe the bottleneck lies somewhere else in the stack. The median in this benchmark with write-mapped databases, with and without DoubleVision, is 1.5 MB/s.

SQLite

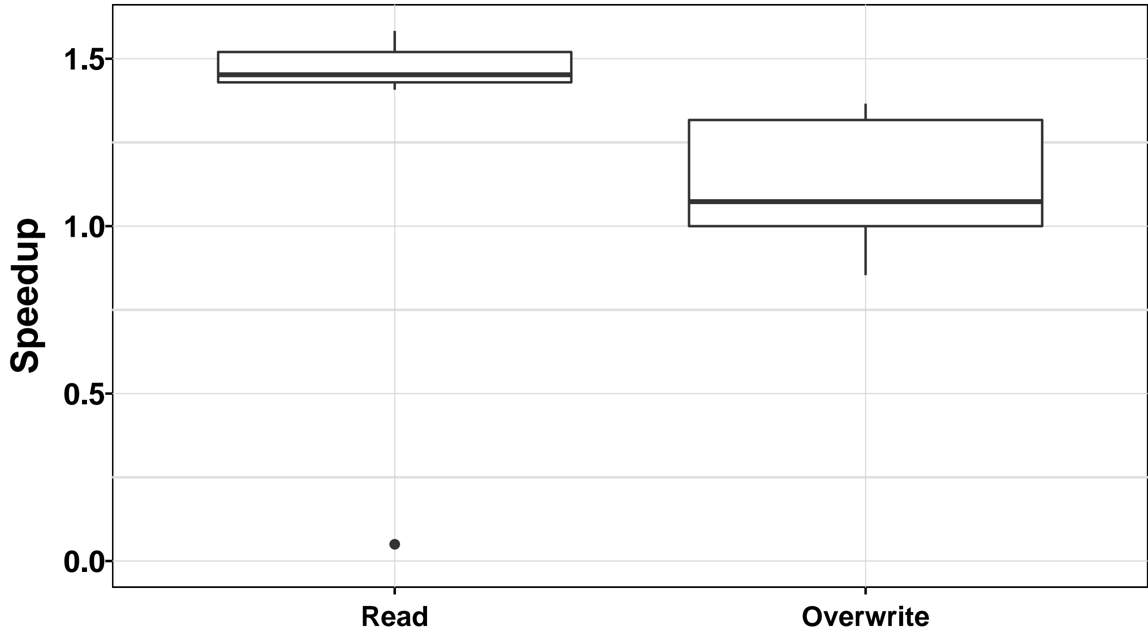

FIGURE 4.9: Performance of DoubleVision SQLite over stock SQLite without memory-mapping the database.

We evaluated the performance of SQLite augmented with DoubleVision using `kvtest`, a benchmark that tests the performance of reading and overwriting blobs [8]. We ran it with and without mapping the database to memory, and in the former configuration, with and without DoubleVision enabled. In Figure 4.9, we compare SQLite with DoubleVision against stock SQLite without memory-mapping the database. As was observed with the IOZone set of benchmarks, using `mmio` increases the performance over `rwio`. In Figure 4.10, we illustrate the slowdown of DoubleVision SQLite and stock SQLite, using memory-mapped databases in both cases.

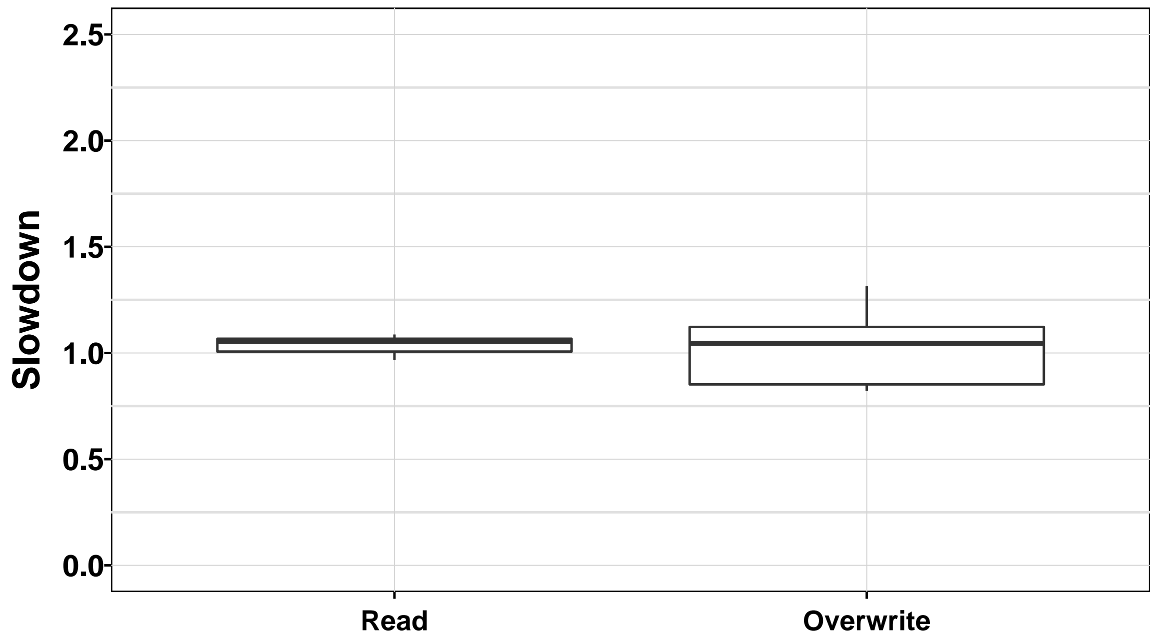

FIGURE 4.10: Slowdown of DoubleVision SQLite over stock SQLite, with memory-mapped databases in both cases.

As expected, there is a slight amount of overhead imposed by DoubleVision. In the overwrite benchmark, DoubleVision performs better than stock in some cases. We believe this is due to I/O variance.

OpenSSL

In this experiment, we compiled the nginx web server from source to use our modified version of OpenSSL. Next, we configured the web server to serve a HTTPS website using 4096-bit RSA keys. We used ApacheBench to perform 1000 SSL handshakes with a concurrency level of 12. In Figure 4.11, we show the slowdown of using DoubleVision over a stock configuration. Note that unlike the previous experiments, we protect the RSA private keys from both stray reads and stray writes. As discussed in Section 4.4, in such cases, we can protect application resources either with memory permissions or parallel memory mappings. We run our experiments with both to see their relative impact. Our results show that DoubleVision imposes less than 5%

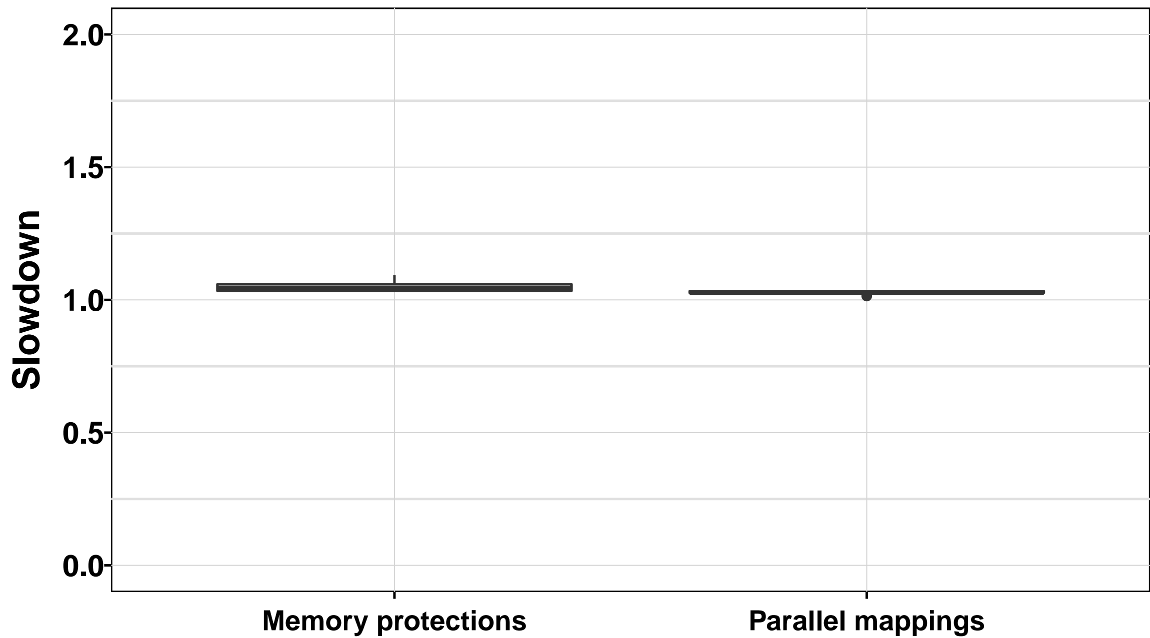

Figure 4.11: Performance of nginx with DoubleVision OpenSSL and stock OpenSSL.

overhead in both cases.

4.5.4 Summary

In summary, the impact of DoubleVision depends on the workload. In numerous cases, I/O overhead eclipses that of DoubleVision. That said, we expect DoubleVision to perform poorly if the workload is CPU-bound and requests constant switching between page tables because the repeated traps would hinder a CPU's core performance.

4.6 Related Work

In Section 4.3, we discussed some prior work while discussing the design of DoubleVision. In this section, we discuss them further and present other related work.

Protecting memory

Unlike DoubleVision, lwCs [46], Wedge [12], Shreds [18], and ERIM [65], were developed with a focus on adversarial scenarios. Hence, the natural question to raise is if simplified versions of these prior work can substitute DoubleVision.

lwCs and Wedge's primitives are inherently non-trivial because they propose primitives that are significantly different from what an OS already provides. It is difficult to simplify them for our present purposes. In the case of Shreds and ERIM, simplifying them would mean just directly using ARM domains or Intel MPK, since much of their contribution is in handling adversarial cases. ARM memory domains is coarser than page protections and is applied in 1 MB granularity, as opposed to the typical page size of 4 KB. Intel MPK is more suitable and if available, may be used in lieu of DoubleVision. That said, as we show in Section 4.4, DoubleVision's use of page tables allows parallel memory mappings, a functionality that may not be implemented with either ARM memory domains or Intel MPK.

Stray memory accesses

The issue of stray memory accesses has been studied in other settings. SafeNVM [45] tries to prevent stray memory writes to portions of a process's AS that is mapped to non-volatile memory (NVM). Similar to us, they identify the need for thread-level memory protections so that only the threads that need to write to the NVM have the appropriate access, while all other threads trap. They propose achieving this with a series of invasive modifications to the CPU architecture, such as new CPU instructions, new thread-local structures, changes to the TLB, and changes to the MMU's access control check sequence. DoubleVision achieves thread-level memory protection without the need for such invasive hardware features and will also be able to prevent stray writes to NVMs mapped in the AS.

Border Control [51] attempts to address this issue from the perspective of pro-

cesses that share their ASes with hardware accelerators such as GPGPUs. Since these hardware accelerators typically access host memory directly through their physical addresses, they are not subject to virtual memory protections. This enables the accelerators to intentionally or accidentally access all of the process's AS and violate confidentiality or integrity. They propose a hardware-based solution that caches the host process's virtual memory mappings and page protections, and enforces them when the accelerators access the host's physical memory. DoubleVision and Border Control are orthogonal to each other.

Memory protection schemes

In PMBD, Chen et al. propose a block device driver to expose persistent memory (PM) to applications [16]. They will use PM via the kernel's `read()` / `write()` interface, and the driver will convert these to simple loads and stores. Since the user-space application cannot directly access persistent memory with raw pointers, the driver has to protect PM only from kernel-level stray accesses. This is done by manipulating the page tables to map the PM to kernel space on a per-core basis, for the duration of the PM load or store. DoubleVision allows applications to protect arbitrary objects in memory without requiring them to change code that does loads and store to use `read()` / `write()`. Moreover, in theory, the API in Table 4.1 can also protect PM, even from kernel-level stray accesses. We leave the application of DoubleVision for PM, protection against kernel-level stray accesses, and a detailed comparison with PMBD to future work.

The Rio file cache transforms RAM into persistent memory that can survive OS crashes [17]. In doing so, the authors propose a variety of techniques to ensure stray kernel-level memory accesses during a crash from corrupting the file cache. Unlike DoubleVision, Rio does not handle stray user-level accesses [48]. RVM enables applications to map regions of memory with transactional support into its virtual

address space [57]. RVM provides a transaction API and maintains a log of events. While corruptions due to stray writes are possible, the log enables data in the RVM to be reverted to a pre-corrupted state.

Others

Gerofi et al. have studied the cost of TLB shootdowns in manycore machines that run parallel workloads [35]. In the regular case, a process's threads share the same page table. When threads in a process change the virtual-to-physical mappings, the kernel has to perform TLB shootdown in all of the CPU cores. However, much of these shootdowns may be unnecessary since threads divide work in parallel workloads, and each thread likely accesses an independent sliver of the shared AS. The authors propose using per-core *partially separated page tables* within an AS to minimize unnecessary TLB shootdowns. Each core has a separate first level page table (`pgd`) for the process but they share the intermediate tables where possible, thus reducing the cost of maintaining multiple tables. In DoubleVision, we use multiple tables per process for parallel memory permissions. That said, the page table optimization we describe in Section 2.1.3 are akin to partially separated page tables, and it is used in our work and theirs to minimize the memory consumption of the additional page tables.

4.7 Conclusion

In this work, we have presented DoubleVision, a system to prevent stray accesses to memory-mapped application resources caused by bugs. DoubleVision uses parallel memory permissions to allow multi-threaded processes to have different permissions to the same address space, allowing a thread to only request access to the resource when it requires it. Otherwise, it will run without permissions to the resource and its stray accesses to the resource will fail. Our experiments show that the performance

impact of DoubleVision is workload dependent, with I/O overhead often eclipsing that of DoubleVision.

5

Conclusion

In this dissertation, we have presented parallel memory permissions, a technique that allows threads to run in parallel while having different permissions to a shared AS. Parallel memory permissions works with commodity hardware that support virtual memory with page tables.

In Chapter 3, we demonstrated an implementation of parallel memory permissions for the Galaxy Nexus smartphone device, running Android and the Linux kernel v3.0.31. We implemented and evaluated it using both ARMv7 memory domains and multiple page tables. In Chapter 4, we demonstrated an implementation for the Raspberry Pi 3, running Raspbian, a debian-based Linux distribution, and the Linux kernel v4.9.61, using just multiple page tables. This supports our case that parallel memory permissions are widely applicable; we have implemented it for both old and new versions of the Linux kernel, and we have it working for different hardware types.

Parallel memory permissions are a good fit for systems that need to run different threads in different protection domains. In SandTrap, we need untracked threads to trap when they access tracked data and in DoubleVision, we need threads to trap if they unintentionally access memory-mapped resources due to software bugs.

5.1 Future work

There are a number of directions in which parallel memory permissions can be extended.

Supporting other ISAs

In both SandTrap and DoubleVision, we implemented parallel memory permissions for the 32-bit ARM architecture. To further demonstrate its generality, it may be implemented for other ISAs, such as x86 and 64-bit ARM. The former would allow us to evaluate it with typical desktop applications while the latter would enable us to port SandTrap to the newer versions of Android that work only with 64-bit ARM CPUs.

Protecting IoT devices

Ho et al. used on-demand DIFT with page protections to track network data and prevent it from being executed [39]. Their intuition is that malware that exploits vulnerabilities in applications running in a system have to be sent over the network, and by preventing data that arrived from the network from being executed, one can prevent malware from being executed. As we discussed in Section 3.2.2, their use of page protections alone means that they are limited to single-threaded processes.

With parallel memory permissions, we can revisit the above work in the context of IoT devices. We make two observations. First, it is not unusual for IoT-devices to have multi-core CPUs. Recall that the Raspberry Pi 3, an IoT-class device, has a quad-core CPU. Second, IoT devices have an operating model that makes it attractive to adversaries. They are typically designed to run fixed tasks and their software may not be updated often, either because service providers do not do so or because users find it time-consuming. If we can successfully use on-demand DIFT on a per-thread basis to track network data and prevent them from being executed,

we can provide a system that prevents malware from exploiting IoT devices that have trusted but potentially buggy and vulnerable software. As with SandTrap, we require parallel memory permissions as a key building block.

Isolating DOM contexts

In the current web browser security model, all JavaScript scripts running on a page have access to the Document Object Model (DOM), a tree-representation of the elements in the web page. At present, it is not possible to classify different scripts with different trust levels, and prevent scripts that are not fully trusted from accessing parts of the DOM. Scripts can only be fully trusted or not at all and in the latter, scripts are prevented from executing.

We believe that parallel memory permissions can provide an alternative security model. By integrating it into the browser, we can execute different scripts with different access permissions. For instance, we can allow a script served by an advertisement network to execute, but prevent it from accessing DOM elements that contain sensitive user data (such as a password field).

Bibliography

[1] Android Developers - Keeping Your App Responsive. https://developer.android.com/training/articles/perf-anr.html.

[2] Capstone — The Ultimate Disassembler. http://www.capstone-engine.org.

[3] IOZone Filesystem Benchmark. http://iozone.org.

[4] LevelDB. http://leveldb.org.

[5] Memory protection keys. https://lwn.net/Articles/643797/.

[6] PIE — instruction encoder / decoder. https://github.com/beehive-lab/pie.

[7] Raspbian. https://www.raspberrypi.org/downloads/raspbian/.

[8] SQLite - kvtest.c. https://www.sqlite.org/src/file/test/kvtest.c.

[9] Appel, A., and Li, K. Virtual Memory Primitives for User Programs. In *Proceedings of ASPLOS '91* (April 1991).

[10] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., le Traon, Y., Octeau, D., and McDaniel, P. FlowDroid: Precice Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of PLDI '14* (June 2014).

[11] Belay, A., Bittau, A., Mashtizadeh, A., Terei, D., Mazières, D., and Kozyrakis, C. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of OSDI '12* (October 2012).

[12] Bittau, A., Marchenko, P., Handley, M., and Karp, B. Wedge: Splitting Applications into Reduced-privilege Compartments. In *Proceedings of NSDI '08* (April 2008).

[13] BoltDB. An embedded key/value database for Go. https://github.com/boltdb/bolt.

[14] BoltDB. BoltDB write performance is not good. https://github.com/boltdb/bolt/issues/237.

[15] Bosman, E., Slowinska, A., and Bos, H. Minemu: The World's Fastest Taint Tracker. In *Proceedings of RAID '11* (September 2011).

[16] Chen, F., Mesnier, M. P., and Hahn, S. A Protected Block Device for Persistent Memory. In *Proceedings of MSST '14* (June 2014).

[17] Chen, P. M., Ng, W. T., Chandra, S., Aycock, C., Rajamani, G., and Lowell, D. The Rio File Cache: Surviving Operating System Crashes. In *Proceedings of ASPLOS '96* (October 1996).

[18] Chen, Y., Reymondjohnson, S., Sun, Z., and Lu, L. Shreds: Fine-Grained Execution Units with Private Memory. In *Proceedings of IEEE SP '16* (May 2016).

[19] Choi, J., Kim, J., and Han, H. Efficient Memory Mapped File I/O for In-Memory File Systems. In *Proceedings of HotStorage '17* (July 2017).

[20] Clause, J., Li, W., and Orso, A. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of ISSTA '07* (July 2007).

[21] Common Weakness Enumeration. CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer. http://cwe.mitre.org/data/definitions/119.html.

[22] Common Weakness Enumeration. CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow'). http://cwe.mitre.org/data/definitions/120.html.

[23] Common Weakness Enumeration. CWE-125: Out-of-bounds Read. http://cwe.mitre.org/data/definitions/125.html.

[24] Common Weakness Enumeration. CWE-415: Double Free. http://cwe.mitre.org/data/definitions/415.html.

[25] Common Weakness Enumeration. CWE-416: Use After Free. http://cwe.mitre.org/data/definitions/416.html.

[26] Common Weakness Enumeration. CWE-787: Out-of-bounds Write. http://cwe.mitre.org/data/definitions/787.html.

[27] Common Weakness Enumeration. CWE-824: Access of Uninitialized Pointer. http://cwe.mitre.org/data/definitions/824.html.

[28] Cox, L. P., Gilbert, P., Lawler, G., Pistol, V., Razeen, A., Wu, B., and Cheemalapati, S. SpanDex: Secure Password Tracking for Android. In *Proceedings of USENIX Security '14* (August 2014).

[29] Denning, P. J. Virtual memory. *ACM Comput. Surv. 2*, 3 (Sept. 1970), 153–189.

[30] Dong, X., Dwarkadas, S., and Cox, A. L. Shared Address Translation Revisited. In *Proceedings of EuroSys '16* (April 2016).

[31] Efstathopoulos, P., Krohn, M., VanDeBogart, S., Frey, C., Ziegler, D., Kohler, E., Mazières, D., Kaashoek, F., and Morris, R. Labels and Event Processes in the Asbestos Operating System. In *Proceedings of SOSP '05* (October 2005).

[32] Egele, M., Kruegel, C., Kirda, E., and Vigna, G. Pios: Detecting privacy leaks in ios applications. In *NDSS* (2011), The Internet Society.

[33] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. TaintDroid: An Information-Flow Tracking system for Realtime Privacy Monitoring on Smartphones. In *Proceedings of OSDI '10* (October 2010).

[34] Frisby, W., Moench, B., Recht, B., and Ristenpart, T. Security Analysis of Smartphone Point-of-Sale Systems. In *Proceedings of WOOT '12* (August 2012).

[35] Gerofi, B., Shimada, A., Hori, A., and Ishikawa, Y. Partially Separated Page Tables for Efficient Operating System Assisted Hierarchical Memory Management on Heterogeneous Architectures. In *Proceedings of CCGrid '13* (May 2013).

[36] Gilbert, P., Jung, J., Lee, K., Qin, H., Sharkey, D., Sheth, A., and Cox, L. P. YouProve: Authenticity and Fidelity in Mobile Sensing . In *Proceedings of SenSys '11* (November 2011).

[37] Gorgovan, C., d'Antras, A., and Luján, M. Mambo: A low-overhead dynamic binary modification tool for arm. *ACM Trans. Archit. Code Optim. 13*, 1 (Apr. 2016), 14:1–14:26.

[38] Greathouse, J. L., Xin, H., Luo, Y., and Austin, T. A Case for Unlimited Watchpoints. In *Proceedings of ASPLOS '12* (March 2012).

[39] Ho, A., Fetterman, M., Clark, C., Warfield, A., and Hand, S. Practical Taint-Based Protection using Demand Emulation. In *Proceedings of EuroSys '06* (April 2006).

[40] Howard Chu. Fork of LevelDB benchmarks. `https://github.com/hyc/leveldb/tree/benches`.

[41] Hsu, T. C.-H., Hoffman, K., Eugster, P., and Payer, M. Enforcing Lease Privilege Memory Views for Multithreaded Applications. In *Proceedings of cCS '16* (October 2016).

[42] Jurriaan Bremer. darm - Efficient ARMv7 Disassembler. `https://github.com/jbremer/darm`.

[43] Kang, M. G., McCamant, S., Poosankam, P., and Song, D. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *Proceedings of NDSS '11* (February 2011).

[44] Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M. F., Kohler, E., and Morris, R. Information Flow Control for Standard OS Abstractions. In *Proceedings of SOSP '07* (October 2007).

[45] Kumar, P., and Huang, H. H. SafeNVM: A Non-Volatile Memory Store with Thread-Level Page Protection. In *Proceedings of IEEE Big Data Conference '17* (June 2017).

[46] Litton, J., Vahldiek-Oberwagner, A., Elnikety, E., Garg, D., Bhattacharjee, B., and Druschel, P. Light-weight Contexts: An OS Abstraction for Safety and Performance. In *Proceedings of OSDI '16* (November 2016).

[47] LMDB. Lightning Memory-Mapped Database Manager. `http://www.lmdb.tech/doc/`.

[48] Lowell, D. E., and Chen, P. M. Free Transactions with Rio Vista. In *Proceedings of SOSP '97* (October 1997).

[49] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of PLDI '05* (June 2005).

[50] Newsome, J., and Song, D. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of NDSS '05* (February 2005).

[51] Olson, L. E., Power, J., Hill, M. D., and Wood, D. A. Border Control: Sandboxing Accelerators. In *Proceedings of MICRO '15* (October 2015).

[52] Qian, C., Luo, X., Shao, Y., and Chan, A. T. On Tracking Information Flows through JNI in Android Applications. In *Proceedings of DSN '14* (June 2014).

[53] Qin, F., Wang, C., Li, Z., Kim, H.-s., Zhou, Y., and Wu, Y. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *Proceedings of MICRO '06* (December 2006).

[54] Quinn, A., Devecsery, D., Chen, P. M., and Flinn, J. JetStream: Cluster-Scale Parallelization of Information Flow Queries. In *Proceedings of OSDI '16* (November 2016).

[55] Razeen, A., Pistol, V., Meijer, A., and Cox, L. P. Better performance through thread-local emulation. In *Proceedings of HotMobile '16* (February 2016).

[56] Rinard, M., Cadar, C., Dumitran, D., Roy, D. M., Leu, T., and William S. Beebee, J. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings of OSDI '04* (December 2004).

[57] Satyanarayanan, M., Mashburn, H. H., and Kumar, P. Lightweight Recoverable Virtual Memory. In *Proceedings of SOSP '93* (August 1993).

[58] Shen, H., Balasubramanian, A., LaMarca, A., and Wetherall, D. Enhancing Mobile Apps To Use Sensor Hubs Without Programmer Effort. In *Proceedings of UbiComp '15* (September 2015).

[59] Spahn, R., Bell, J., Lee, M. Z., Bhamidipati, S., Geambasu, R., and Kaiser, G. Pebbles: Fine-Grained Data Management Abstractions for Modern Operating Systems. In *Proceedings of OSDI '14* (October 2014).

[60] SQLite. Memory-Mapped I/O. `https://sqlite.org/mmap.html`.

[61] Sun, M., Wei, T., and Lui, J. C. TaintART: A Practical Multi-level Information-Flow Tracking System for Android RunTime. In *Proceedings of CCS '16* (October 2016).

[62] Symas. In-Memory Microbenchmark. `http://www.lmdb.tech/bench/inmem/`.

[63] Tang, Y., Ames, P., Bhamidipati, S., Bijlani, A., Geambasu, R., and Sarda, N. Clean OS: Limiting Mobile Data Exposure with Idle Eviction. In *Proceedings of OSDI '12* (August 2012).

[64] Tarasov, V., Bhanage, S., Zadok, E., and Seltzer, M. Benchmarking File System Benchmarking: It *IS* Rocket Science. In *Proceedings of HotOS '11* (May 2011).

[65] Vahldiek-Oberwagner, A., Elnikety, E., Garg, D., and Druschel, P. Erim: Secure and efficient in-process isolation with memory protection keys. arXiv:1801.06822, 2018.

[66] Witchel, E., Cates, J., and Asanović, K. Mondrian Memory Protection. In *Proceedings of ASPLOS '02* (October 2002).

[67] Woodruff, J., Watson, R. N., Chisnall, D., Moore, S. W., Anderson, J., Davis, B., Laurie, B., Neumann, P. G., Norton, R., and Roe, M. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceedings of ISCA '14* (June 2014).

[68] Yan, L. K., and Yin, H. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proceedings of USENIX Security '12* (August 2012).

[69] Zeldovich, N., Boyd-Wickizer, S., Kohler, E., and Mazières, D. Making Information Flow Explicit in HiStar. In *Proceedings of OSDI '06* (November 2006).

[70] Zhou, Y., Wang, X., Chen, Y., and Wang, Z. ARMlock: Hardware-based Fault Isolation for ARM. In *Proceedings of '14* (November 2014).

Biography

Ali Razeen was born on the 11th of December, 1986 in Singapore. He earned his BComp (Computer Science) from the National University of Singapore in 2011 and his Ph.D. in Computer Science from Duke University in 2018. While at Duke, he has published in top Computer Science conferences such as USENIX Security and MobiSys. He has also been recognized for his contributions to teaching and won an Outstanding TA award from the Computer Science department in 2017.