SandTrap: Tracking Information Flows On Demand with Parallel Permissions

Ali Razeen
Duke University

Alvin R. Lebeck
Duke University

David H. Liu
Princeton University

Alexander Meijer
Duke University

Valentin Pistol
Duke University

Landon P. Cox
Duke University

ABSTRACT

The most promising way to improve the performance of dynamic information-flow tracking (DIFT) for machine code is to only track instructions when they process tainted data. Unfortunately, prior approaches to on-demand DIFT are a poor match for modern mobile platforms that rely heavily on parallelism to provide good interactivity in the face of computationally intensive tasks like image processing. The main shortcoming of these prior efforts is that they cannot support an arbitrary mix of parallel threads due to the limitations of page protections.

In this paper, we identify parallel permissions as a key requirement for multithreaded, on-demand native DIFT, and we describe the design and implementation of a system called SandTrap that embodies this approach. Using our prototype implementation, we demonstrate that SandTrap's native DIFT overhead is proportional to the amount of tainted data that native code processes. For example, in the photo-sharing app Instagram, SandTrap's performance is close to baseline (1x) when the app does not access tainted data. When it does, SandTrap imposes a slowdown comparable to prior DIFT systems (~8x).

CCS CONCEPTS

• Security and privacy → Information flow control; **• Human-centered computing → Mobile computing**; **• Software and its engineering → Virtual memory**; **Multithreading**;

KEYWORDS

parallel memory permissions, dynamic information-flow tracking, native code

ACM Reference Format:
Ali Razeen, Alvin R. Lebeck, David H. Liu, Alexander Meijer, Valentin Pistol, and Landon P. Cox. 2018. SandTrap: Tracking Information Flows On Demand with Parallel Permissions. In *MobiSys '18: The 16th Annual International Conference on Mobile Systems, Applications, and Services, June 10–15, 2018, Munich, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3210240.3210321

MobiSys '18, June 10–15, 2018, Munich, Germany

ACM ISBN 978-1-4503-5720-3/18/06…$15.00
https://doi.org/10.1145/3210240.3210321

1 INTRODUCTION

Dynamic information-flow tracking (DIFT or taint-tracking) is an important building block for many experimental mobile services [10, 16, 17, 31, 32, 34]. The most popular DIFT implementation for mobile is TaintDroid [15], which does not precisely track flows through native code despite apps' increasing reliance on native third-party libraries. Furthermore, extending DIFT to native code is imperative for monitoring the growing class of apps that use native libraries to process sensor data, perform computer vision, and support augmented reality.

Tracking information flows through native code typically requires instruction-level emulation. When a thread is tracked, the system intercepts each instruction, updates its DIFT state using a taint-propagation logic, and then executes the original instruction. Unfortunately, tracking a thread can degrade its performance by a factor of ten or more, which makes continuous native tracking impractical for most services. Prior work on *on-demand DIFT* [20, 27] attempts to reduce the impact of tracking threads by only emulating code when it handles tainted data. Under on-demand DIFT, untracked threads that avoid accessing tainted data run at full speed.

On-demand DIFT relies on page protections to safely transition a thread from untracked to tracked execution. Before an untracked thread runs, the system places no-access permissions on pages holding tainted data to trigger a fault if an untracked thread accesses those pages. If an untracked thread accesses a tainted page, the fault handler removes protections on tainted pages to avoid more traps during emulation. However, this approach to on-demand DIFT limits parallelism because tracked and untracked threads cannot run at the same time under different virtual-memory protections.

This limitation is particularly painful for Android apps, which often consist of a main UI thread running managed code and background threads that perform computationally intensive tasks using native libraries. Background threads communicate with the main UI thread through shared buffers, and the main thread's managed runtime (e.g., Dalvik) must access those buffers without triggering a fault and launching emulation. Thus, under existing approaches to on-demand DIFT, untracked background threads (i.e., the threads that would most benefit from on-demand DIFT) cannot run in parallel with threads executing managed code (e.g., the main UI thread) or with tracked background threads.

In this paper we present the design, implementation, and evaluation of a native DIFT system for Android called *SandTrap* that addresses the shortcomings of prior approaches to on-demand DIFT. The key observation underlying the design of SandTrap is that *parallel permissions* are a crucial requirement for multithreaded,

on-demand native DIFT of mobile apps. Parallel permissions allow threads in the same address space to execute in parallel under different page protections.

We developed two implementations of parallel permissions in SandTrap: (i) by using a little-used feature of ARM processors called *memory domains*, and (ii) by maintaining two page-tables per app. The first implementation demonstrates how special hardware primitives can be used to implement parallel permissions, and the second implementation demonstrates the generality of SandTrap. In a previous position paper [30], we speculated that ARM memory domains could be a useful mechanism for making native DIFT overhead proportional to the amount of tainted data accessed by native code. In this paper, we present the design and implementation of a complete system that achieves our goal through memory domains and two page-tables.

This paper makes the following contributions:

- We identify parallel permissions as a key requirement for multithreaded, on-demand native DIFT for mobile apps. Parallel permissions allow native code that does not access tainted data to run at full speed in parallel with managed threads like the main UI thread and tracked background threads.
- Using our SandTrap implementation to track microphone data through an Instagram workload that accesses camera data, we show that SandTrap's native-code slowdown is close to a baseline of stock Android. In addition, because untracked threads run at full speed, Instagram under SandTrap consumes only 10% more energy than stock Android. The same Instagram workload under continuous native DIFT suffers a native-code slowdown of approximately 8x while consuming 44% more energy compared to stock Android.

2 BACKGROUND

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

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 multithreaded, 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, it 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.

2.2 Dynamic Information-flow Tracking

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 *taint 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 tainted 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 [13, 23, 38]. 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.

TaintDroid. TaintDroid [15] is a widely used DIFT system for Android that provides a good balance of performance and precision 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 [17], data deletion [32], cache eviction [34], energy conservation [31], and password management [10]. TaintDroid has also been instrumental for studying how mobile apps handle sensitive information [15, 16].

Despite its widespread use, TaintDroid does not precisely track flows through native code. TaintDroid does not even allow apps to load native third-party libraries, and it provides imprecise, method-level tracking for native platform libraries. Disallowing native third-party libraries is an increasingly problematic limitation. We collected information about 86,000 apps from the Google Play Store in March of 2016, each with more than 100,000 downloads, and found that over 43% used at least one native third-party library. This is a substantial increase from the 5% reported in the TaintDroid paper from 2010. Furthermore, we have observed many flows through native third-party 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.

Native DIFT. 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. For TaintDroid, 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. Performing DIFT on a bytecode requires just a few additional instructions: loads for the source objects' labels, an 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% [33].

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 [37] and NDroid [26] 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 [9, 25].

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 [14, 22]. These issues are particularly challenging with native code, and the popular static information-flow tracking tool FlowDroid [4] does not support native code.

Minemu [7] 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 [28] accelerated DIFT for replayed processes by parallelizing the work across a compute cluster, but this approach is limited to replayed processes. Finally, recent work on optimistic hybrid analysis [11] uses a combination of unsound static analysis and speculative invariant checks at runtime to reduce the amount of DIFT emulation that must be performed. While a promising way to reduce DIFT overhead, optimistic hybrid analysis requires access to library source code whereas native third-party libraries are shipped as binaries.

On-demand native DIFT. The most promising approach to improving native DIFT performance on Android is to track on-demand by only emulating code when it accesses tainted data [20, 27, 30]. This approach can lead to substantial performance gains when accessing tainted data is rare. For example, if a system tracks location or microphone data, native image-processing libraries should run at full speed. However, in the worst case an app may spend *all* of its time handling tainted data and its performance will be the same as continuous DIFT.

Xen [20] uses page protections to track on-demand by setting no-access permissions on pages holding tainted data. As long as a virtual machine (VM) avoids tainted data, it runs at full speed. 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 tainted pages during emulation so that instrumented code can access tainted data and label storage without faulting. If a tracked VM runs long enough without tainted data in its registers, the hypervisor can stop tracking the VM by restoring protections on tainted pages and running the VM at full speed.

LIFT [27] avoids unnecessary emulation by performing live-in and live-out analysis of code segments to identify when a segment

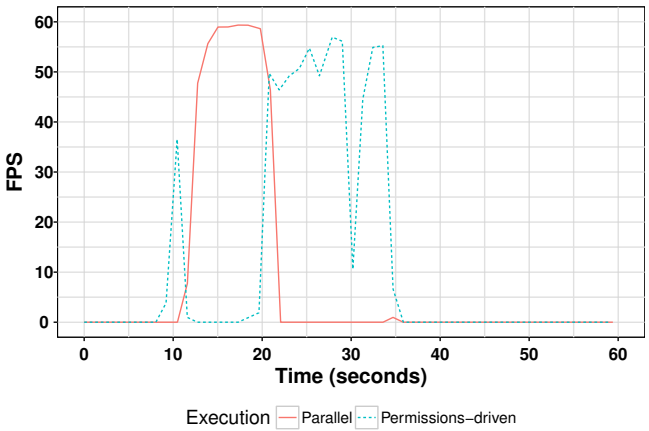


Figure 1: The frame rate recorded from a dual-core Android smartphone when loading a PDF file in eBooka, an ebook reader app, with parallel and permissions-driven scheduling.

can run at full speed. Segments with untainted inputs and outputs do not need to be emulated, and any code segments that could read or overwrite tainted data must be emulated. Like Xen, LIFT uses page protections to prevent untracked code from accessing label storage and removes those protections during emulation.

Multithreaded on-demand DIFT. By default, page protections must be the same for all threads executing in an address space. As a result, as soon as Xen and LIFT start to track a thread they can no longer execute untracked threads on other cores. More generally, requiring different protections for tracked and untracked threads, while using a protection model that allows one set of active protections, limits parallelism; tracked threads cannot run with protections enabled, and untracked threads cannot trap with protections disabled. On a single-core machine, on-demand DIFT systems like LIFT or Xen could restore the appropriate protections when switching the processor between tracked and untracked threads. But on multicore processors app performance would suffer because tracked and untracked threads could not run in parallel.

If an on-demand DIFT system runs on a multicore machine and relies on the default page-protections behavior, it must do one of two things when an untracked thread accesses tainted data and begins emulation: (i) conservatively emulate all executing threads in the process to identify and track future accesses to tainted data, or (ii) use *permissions-driven* scheduling so that threads that require different page protections do not execute in parallel. The first approach forces threads that would otherwise be untracked to pay a significant performance penalty, and the second approach limits parallelism.

2.3 Case study: eBooka

We demonstrate how permissions-driven scheduling can limit parallelism with the eBooka PDF reader. 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 ten 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. In this experiment, eBooka's background threads are untracked and run with tainted pages marked no-access. eBooka's main UI thread must run with page protections disabled to prevent the managed-code runtime from triggering a fault.

We ran the experiment on a dual-core Galaxy Nexus smartphone using permissions-driven scheduling and an unmodified Linux scheduler. The unmodified *parallel* scheduler runs all eBooka threads in parallel. To implement permissions-driven scheduling, we modify the Linux scheduler so that it does not execute threads requiring different page protections at the same time. Our results are in Figure 1.

In both the parallel and permissions-driven cases, the frame rate starts out at zero 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 waiting period. At this point the app displays a "loading" message and the loading circle continuously spins. At time $t = 22$ the app loads the file and displays the first page. Because there are no more updates to the UI, the frame rate drops back to zero FPS.

Under permissions-driven scheduling, the main UI thread (running managed code) cannot run in parallel with untracked background threads. Depending on which thread is scheduled, either the UI will become unresponsive (when the untracked threads run) or loading and parsing the PDF will stall (when the UI thread runs). Both effects are observable in Figure 1.

After the initial wait period, the app displays the loading bar just before the frame rate drops to zero FPS for about eight seconds. During this interval, the UI thread pauses while the untracked background threads run. Any input events from the user during this period would not be handled within Android's five-second deadline, and would cause an ANR dialog to appear [2]. Between $t = 20$ and $t = 33$ the UI stutters as the UI thread and the background threads contend for the CPU. Eventually, at $t = 35$ the app loads the file and the frame rate drops back to zero FPS because there are no more UI updates. As expected, permissions-driven scheduling limits parallelism, leading to poor interactivity and slow load times.

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

3 SYSTEM OVERVIEW

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

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 to tracking managed code and native libraries in Android is to use the same instruction-level mechanism 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.

However, applying a single low-level DIFT mechanism to all execution environments makes understanding managed-code behavior 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 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%.

SandTrap tracks managed code and native third-party libraries separately, relying on TaintDroid to continuously track managed code and using on-demand DIFT for native third-party 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, 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 Dalvik 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 [18] to instrument native third-party 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 native third-party code under MAMBO instrumentation. An untracked thread executes native third-party 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.

Avoid permissions-driven scheduling. We initially thought that tracking managed code and native code separately would mitigate most of the problems with permissions-driven scheduling, 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 scheduling. 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 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 tainted 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 tainted 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 scheduling, untracked and managed threads cannot run in parallel when a shared buffer resides on a page containing tainted data. As demonstrated earlier, this restriction can cause poor interactivity and a possible ANR dialog because the main thread has to pause whenever untracked threads execute.

To avoid stalling the main thread because of permissions-driven scheduling, SandTrap provides *parallel permissions*, which allows managed threads (including the main thread) to safely run in parallel with tracked *and* untracked threads.

Take advantage of hardware primitives. One way to provide parallel permissions is to map physical pages holding tracked data into two virtual pages, to apply different permissions to the two virtual pages, and to redirect memory accesses depending on the current thread's status [3]. 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 tainted data. Thus, SandTrap cannot rely on software protections alone. When possible SandTrap uses special hardware features to implement parallel permissions, and we have also implemented SandTrap using generic virtual-memory features.

First, the ARMv7 architecture provides a *memory domains* hardware primitive. It is similar to x86 memory protection keys [1] and has been used in several recent systems to protect portions of an address space [8, 39] and to consolidate physical-page use [12].

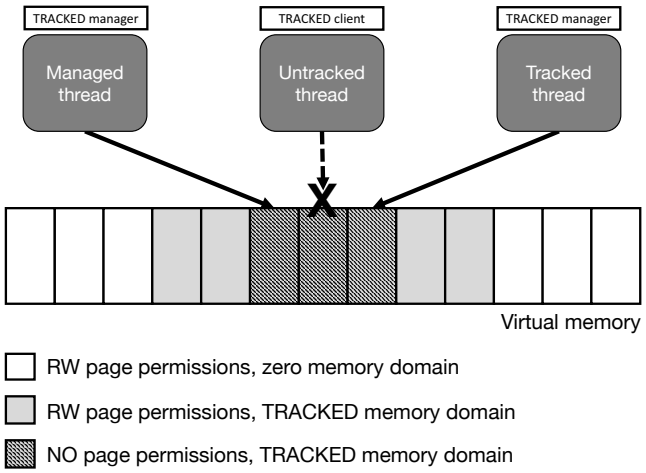


Figure 2: Parallel permissions with ARM memory domains.

To the best of our knowledge, SandTrap is the first system to use memory domains for on-demand emulation.

Memory domains allow the 1MB region of memory covered by a top-level page-table entry (PTE) to be tagged with one of 16 domains. In addition, each 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 no-access mode for a domain, then any attempt to access memory tagged under that domain will fail and trigger a domain fault, regardless of the protections in the address's PTE. If a core is in manager mode for a domain, then any attempt to access memory tagged under that domain will succeed regardless of the PTE protections. Finally, if a core is in client mode for a domain, then attempts to access memory tagged under that domain will be handled according to PTE protections.

SandTrap provides parallel permissions with ARM memory domains by tagging all native pages holding tainted data with a new `TRACKED` domain. Figure 7 shows how SandTrap protects and tags regions of memory holding tainted data. When code copies tainted data to a page, SandTrap ensures the page has no-access permissions so that untracked threads trap if they access the tainted page. Next, SandTrap tags the top-level PTE of the page with the `TRACKED` memory domain. Managed and tracked threads run in manager mode for the `TRACKED` domain, so that they bypass access restrictions placed on tainted pages. At the same time, untracked threads run in client mode for the `TRACKED` domain and abide by the tainted 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.

Unfortunately, ARM has deprecated memory domains for CPUs running in 64-bit mode (i.e., AArch64 mode in ARMv8). For architectures that lack hardware primitives similar to memory domains or memory protection keys, parallel permissions may be implemented by maintaining two synchronized page tables for every running app: *protected* and *unprotected*. The virtual-to-physical mappings of both tables are identical, and the tables differ only in that pages holding tainted data have no-access permissions in the protected table. Under this approach, the kernel is in charge of switching the page table used by a core depending on the state of the executing thread. Untracked threads use the protected page table while managed and tracked threads use the unprotected table.

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 so SandTrap imprecisely tracks three broad classes of platform native libraries.

First, SandTrap does not precisely track Bionic methods on which the SandTrap implementation depends. 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 synchronization, 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 these methods 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 an app sends tainted data over the network, SandTrap logs 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, it sets the output buffer's label to the union of all input labels.

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 [10] 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.

4 SANDTRAP

SandTrap continuously tracks Dex bytecodes using TaintDroid and performs on-demand DIFT on native third-party 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 support parallel permissions so that managed, tracked, and untracked threads can execute in parallel.

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 taint source, such as the GPS, microphone, or camera sensor. 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 300MB region immediately below the process's primary stack region. SandTrap allocates stacks for new threads by mmapping new memory above the heap region. 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 native third-party 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 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 `ReleaseArrayElements`. 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` could be optimized by only synchronizing labels for buffers that native code accesses via `GetDirectBufferAddress`, but 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.

4.2 Parallel permissions

To keep track of which pages hold tainted data, SandTrap maintains a count of the number of tainted words on each page. If a page's count increases from zero to one, then the page must be protected. If the page's count decreases from one to zero, its protections must be removed. Updating page counts and page protections is handled by the JNI layer and the native DIFT implementation in collaboration with kernel code.

Page protections are a relatively coarse-grained protection mechanism, and prior work on unlimited watchpoints has suggested that using pages to trap-and-emulate can lead to high false-trap rates [19]. That is, a thread may trigger emulation by accessing untainted data on a tainted page. Thankfully, this has not been our experience with SandTrap, perhaps because native threads on Android typically perform bulk processing of large buffers.

As previously described, Android's default page-protection scheme is not enough to provide parallel permissions. Thus, depending on the underlying hardware, SandTrap uses page protections in conjunction with ARM memory domains or two page-tables to provide on-demand native DIFT.

ARM memory domains. When hardware memory domains are available, SandTrap defines a new domain called `TRACKED`, and managed and tracked threads run in manager mode for the `TRACKED` domain. Running in manager mode allows these threads to bypass page protections for pages falling under the `TRACKED` domain. However, domains cover coarse 1MB regions, and a thread in manager mode will ignore protections for *all* pages under the `TRACKED` domain, including pages without tainted data. This can be dangerous in systems like Android that use copy-on-write and a shared zero page to improve memory utilization.

By default, virtual pages that map to these shared physical pages are write protected to defer the work of creating a physical copy.

However, if such a virtual page falls under the TRACKED domain, then a thread in manager mode could bypass the page's write protections and modify the underlying physical page. Such errant writes would be visible to all threads with the physical page mapped into their address space. Modifying the zero page is particularly damaging and often leads to a full system crash.

As a result, no physical page mapped into a virtual page in a TRACKED region of memory can be mapped into any other virtual page. SandTrap prevents threads from corrupting shared physical pages by pro-actively allocating and mapping-in unique physical pages for each virtual page marked copy-on-write or mapped to the zero page. In the worst case, a 1MB region of virtual memory could be populated by a single tainted page with the remainder of the region mapped to zero pages. In this case, SandTrap will waste nearly 1MB of physical memory allocating identical zero-filled physical pages. Fortunately in practice, SandTrap typically allocates only a few new physical pages when tagging a TRACKED region. This is because most heap and stack pages that share a first-level PTE with a tainted page are either unmapped or are not marked for copy-on-write.

Similarly, when an app allocates a new anonymous page with mmap under the TRACKED domain, SandTrap immediately allocates a new physical page instead of relying on copy-on-write. Normally the kernel maps anonymous pages to physical-address zero, and an access triggers a translation fault. Manager mode suppresses protection faults, but not translation faults. Thus, if a thread in manager mode accesses an unmapped page in the TRACKED domain, i.e., one with a translation address of zero, ARM will still trigger a translation fault. Avoiding translation faults on anonymous pages helps the kernel identify legitimate translation faults, such as an app giving up a virtual page and the kernel updating the page's PTE with a zero address.

Finally, our current implementation of SandTrap does not allow pages in the TRACKED domain to be shared between processes. This is not a fundamental limitation; to correctly support shared pages, the SandTrap labels of a shared tainted page must be also shared. We leave this for future work, but we have not seen this behavior in practice since Android apps typically use Binder for inter-app communication.

Two page-tables. When memory domains are not available, SandTrap provides parallel permissions by maintaining two page-tables per app. Untracked threads use a *protected* table, which has protections on pages with tainted data, while managed and tracked threads use an *unprotected* table. The main challenge for SandTrap is synchronizing the virtual-to-physical mappings in both tables.

The Linux kernel manages the AS of a process using the mm_struct data structure. It defines metadata fields about the AS, including a pointer to a top-level page-table. Threads within a process use the same mm_struct and page-table instance. If changes need to be made to the AS, the kernel consults the metadata in the mm_struct before updating the page table. For example, a thread may map a new anonymous page at virtual address X with mmap. The kernel will initially map X to the zero page, which is marked read-only. When the thread performs the first write to X, the core will raise a write fault, which will invoke the kernel's fault handler. The kernel will use the metadata in mm_struct to identify that the fault was caused by a write to a mapped but unallocated page. The kernel will allocate a new physical page, update the top-level table so that X maps to this new page, and return from the fault handler. From this point onward, writes to X will succeed without trapping.

Given this starting point, SandTrap designates the page-table pointer already defined in mm_struct as the unprotected page-table and creates a new pointer to the protected page-table. The protected table is lazily allocated when a thread changes to an untracked state, or when tainted pages are protected. In addition, SandTrap ensures that changes to mappings in the unprotected table are propagated to the protected table, while ensuring that page protections are kept intact. This approach significantly simplifies the implementation because changes to mappings propagate in one direction, from the unprotected to the protected table.

The second challenge of maintaining two page-tables is managing the TLB. The TLB caches both virtual-to-physical mappings and page protections. Each TLB entry is tagged with the current address space identifier (ASID). This allows the core to avoid repeated page-table walks on frequent accesses to the same memory locations. When a thread switches from managed or tracked to untracked, SandTrap must ensure that the core does not use previously cached TLB entries or the untracked thread might access tainted data without trapping. One option is to flush the TLB whenever the kernel switches between the unprotected and protected tables in a core, but this will increase overhead and may not be sufficient when TLBs are shared by multiple cores. Instead, SandTrap assigns each app a second ASID when allocating its protected page-table. In these cases, the app has two ASIDs, one for each table, and the kernel sets the appropriate ASID on the core based on the page table used by the thread.

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 native third-party method, SandTrap copies the method arguments and updates the native labels as described in Section 4.1. SandTrap then asks the kernel to change the state of the thread to untracked. Depending on the implementation of parallel permissions, the kernel either updates the DACR or the page table.

Untracked threads run at full speed as long as they do not access a page holding tainted data. If an untracked thread never accesses a tainted page, then when it returns through the JNI layer, SandTrap switches the thread's state to managed before returning control to TaintDroid. However, if an untracked thread accesses a page with tainted 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 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 tainted data, a tracked thread that is already running under DIFT, or, worst of all, a managed thread like the main thread.

On a trap, SandTrap's signal handler switches the thread to a 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 native third-party 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. Thus, when a tracked thread makes an OpenGL call, SandTrap pauses emulation, performs the OpenGL call, and then resumes emulation.

A consequence of SandTrap's memory-protection scheme is that each Dex call into native third-party code requires at least two kernel traps: one to change the thread to untracked mode, and one to change it back to managed mode on return. Upcalls from untracked threads into managed code 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 tainted data. In our experience these moments are fleeting, and threads typically start processing tainted data very shortly 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 through the JNI.

5 EVALUATION

To evaluate our SandTrap prototype we ask the following questions: (1) Does on-demand native DIFT provide better performance than continuous tracking? (2) Does SandTrap incur false traps? If so, are they truly false traps, or does the thread eventually touch tainted data? (3) Is on-demand DIFT more energy-efficient than continuous tracking? (4) How much memory does SandTrap consume? To answer the first two questions, we provide data about only native-method performance, i.e., the results are not end-to-end.

End-to-end timing results require measuring the elapsed time between an initial user interaction and a final app event (typically to update the UI). In Android, a user interaction may spawn many asynchronous events, and accurately accounting for the time to process those events requires a tool such as AppInsight [29] for Android. Lacking such a tool, we rely on energy experiments to characterize the end-to-end impact of SandTrap.

5.1 Experimental methodology

For all of our experiments, we use a dual-core Galaxy Nexus smartphone running Android 4.1.1. We add instrumentation to record timings, basic-block counts, and other data as needed for the specific experiment. Our baseline for comparison is stock Android. For timing measurements, we modify Dalvik to log timestamps at native-method entry and exit. We use the Android Monitor infrastructure to log and retrieve experiment data.

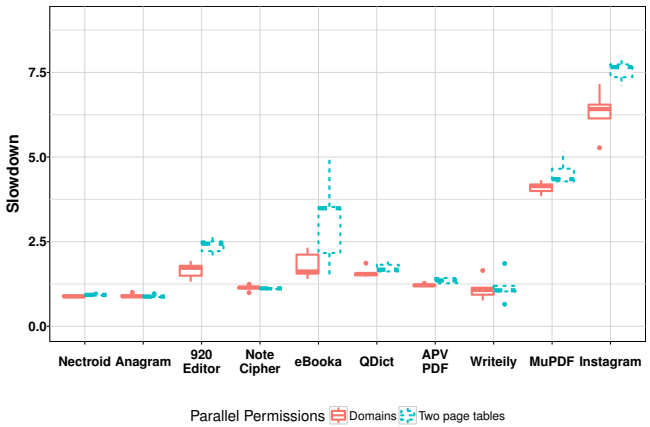


Figure 3: App-study native-code slowdown.

We use ten Android applications representing a variety of use cases [1]. Prior work reports that TaintDroid and TaintART overhead is only 14%, and therefore *we do not report Dalvik-induced overhead and focus on only native overhead.* 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 for all native-method invocations) unless otherwise stated. We note that our performance results represent a worst case overall execution time since we simply sum the times to complete each native-method invocation.

We use Instagram to explore SandTrap's overhead in detail. Results from these experiments are somewhat noisy since they interact with a live Internet service. For example, we find that each Instagram experiment exhibits around 200 native method invocations. The variation is likely due to Instagram authentication activity. To standardize our Instagram analysis, we automate measurements using a Python script executing on a desktop PC to issue commands to the phone over the Android Debug Bridge (ADB). This setup allows us to launch apps, virtually click on the UI, and virtually type inputs, while also collecting instrumentation output from the Android Monitor.

Instagram launches many background threads to apply filters to images and generate thumbnails. Even after an image is uploaded to Instagram servers, the phone could still be processing photo data in the background. Our experiments are designed to capture all Instagram activity for a given post, including trailing post-upload image processing. In particular, our script issues a series of commands instructing Instagram to take a photo from the front camera, annotate the image, and post it.

5.2 On-demand vs continuous DIFT

For our experiments, we use three taint sources: camera, location, and microphone. We measure performance as the total time that all threads spend in native execution, normalized to a baseline of

[1] 920 Editor, Anagram, APV PDF, eBooka, Instagram, MuPDF, Nectroid, NoteCipher, QDict, and Writely

stock Android. Figure 3 shows SandTrap performance for ten apps using boxplots. SandTrap introduces modest slowdowns (1x to 8x). Instagram is the only app that accesses tainted data (camera), and it incurs the largest slowdown. For the other apps, the range of slowdowns reflects different numbers and durations of native calls. Recall that each native call in SandTrap incurs two traps, one to switch the thread from managed to untracked mode on entry and one to switch back to managed mode on exit. In some cases, such as MuPDF, the overhead is due to additional crossings from native code to managed code. As discussed in Section 4.3, each upcall from native to managed code requires two kernel 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.

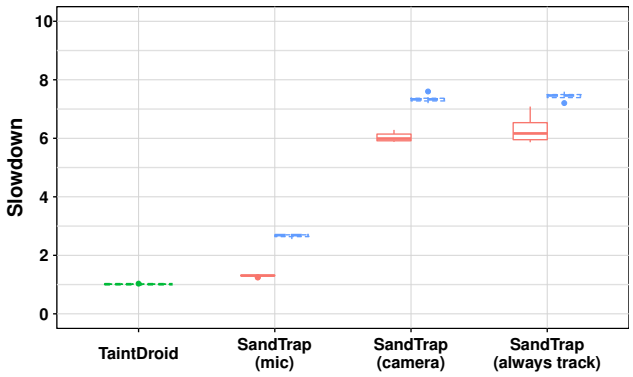
We next examine SandTrap's overall performance using Instagram with different taint sources as a representative benchmark. We specify either the microphone or the camera as a taint source. These cases represent two extremes. Instagram captures pictures with the camera and spends significant time in native routines processing the resulting images. In contrast, when taking a photo, Instagram does not access the microphone, and as a result image-processing background threads should remain untracked. We also include results for unmodified TaintDroid and continuous DIFT.

The baseline stock Android spends approximately 10.5 seconds executing native code and makes approximately 200 native-method invocations. The results, shown in Figure 4a, reveal two important highlights: (1) SandTrap performance approaches the baseline system for use cases that access little or no tainted data, and (2) SandTrap incurs 6-8x slowdown when DIFT is required. These results demonstrate the importance of on-demand DIFT and confirm that SandTrap achieves worst-case performance comparable to other native DIFT implementations.

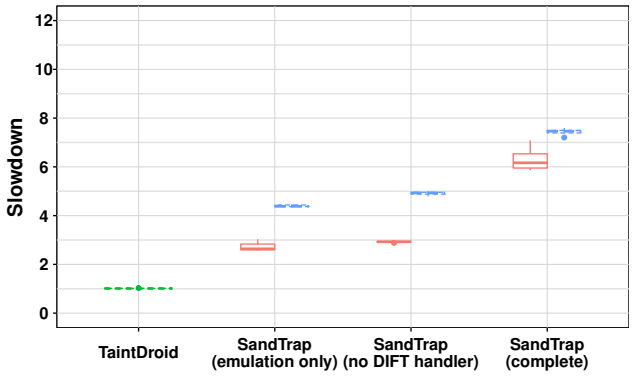
TaintDroid achieves performance equal to stock Android, as expected since we focus only on native-method execution and TaintDroid does not perform DIFT on native code. We observe that for Instagram and tainted microphone data, SandTrap incurs a modest 1.5x slowdown (using domains) due to the overhead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

To better understand SandTrap performance, we next examine the overhead of individual SandTrap components. When a thread accesses a protected page with tainted data, SandTrap transfers control to a MAMBO-based instruction emulator. SandTrap adds two additional overheads for DIFT on each MAMBO basic block beyond baseline MAMBO emulation: (1) writing DIFT rules to a buffer and (2) processing DIFT rules in a 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 three different scenarios that emulate all native methods regardless of whether they access tainted data or not: (1) baseline MAMBO emulation, including trap overhead to begin emulation (emulation only), (2) DIFT rule recording only (no DIFT handler), and (3) continuous tracking (complete). Each step adds an incremental overhead beyond the baseline stock Android. Figure 4b shows the performance of Instagram with tainted camera data for TaintDroid and the three scenarios described above.



(a) Different taint sources



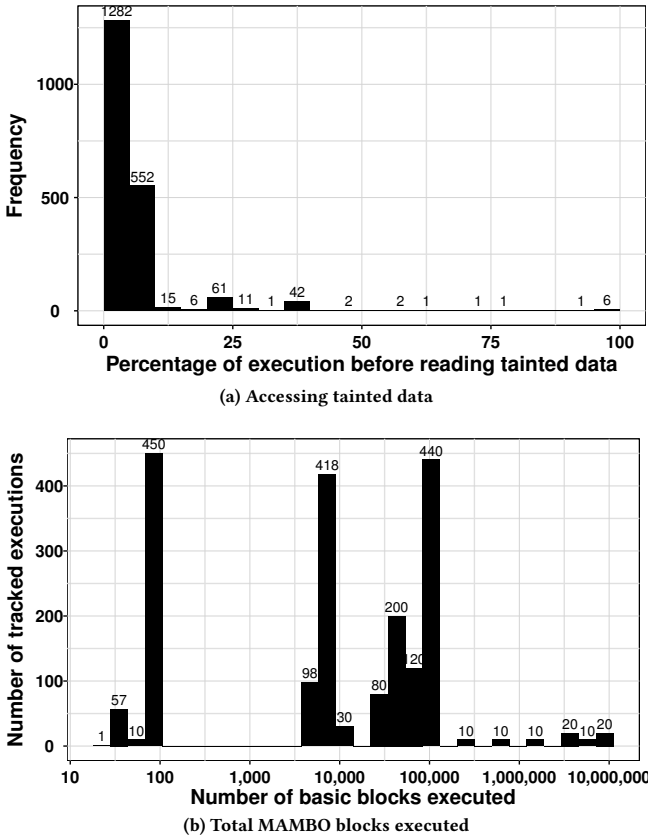
(b) SandTrap component overhead

Figure 4: SandTrap overhead using Instagram.

SandTrap's DIFT rule logging introduces minimal additional overhead beyond instruction emulation. The biggest source of overhead is the DIFT handler that processes DIFT rules at the end of each MAMBO basic block. There may be opportunities to reduce this overhead since we have not optimized the DIFT handler in our implementation.

5.3 False traps

Since SandTrap uses coarse page protections to trap accesses to tainted data, it will also trap native methods that (1) access untainted data that shares a page with tainted data, or (2) overwrite tainted data with untainted data. A false trap occurs if we begin emulation for a native method, but it never accesses tainted data before returning to Dalvik. In some cases, a trap may occur because a native method accesses untainted data, but the method may



(a) Accessing tainted data

(b) Total MAMBO blocks executed

Figure 5: SandTrap false traps using Instagram.

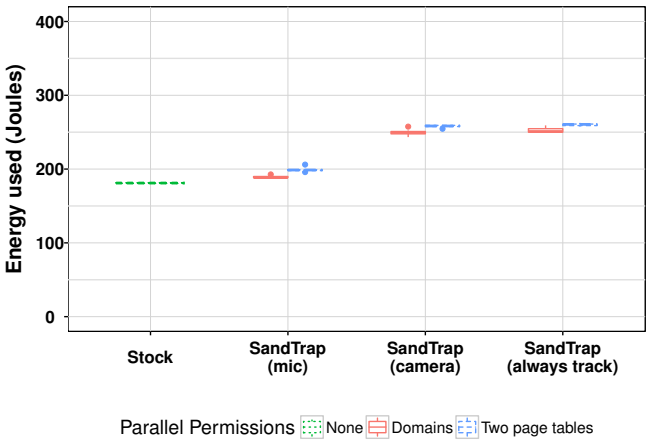


Figure 6: SandTrap energy consumption using Instagram.

eventually access tainted data before returning to Dalvik. In these situations, the native method simply triggers a trap earlier than strictly necessary.

We ran another set of Instagram experiments (with two page-tables) to analyze why native methods trap. In sum, there were a total of 1,996 emulated native-method invocations. Our results reveal that false traps are rare (only 12 invocations), and only a small percentage of methods read tainted data within the first MAMBO block (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 tainted 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 captures, as a percentage, how far a native method executes before it accesses tainted data following a trap.

Figure 5a shows an aggregate histogram where the x-axis is the percentage of execution for each native invocation in dynamic MAMBO blocks, where each bin is 5%. The y-axis counts the instances that read tainted data for a given percentage of execution. From this figure we observe that most native invocations read tainted data within the first five to ten percent of basic blocks following a trap. Specifically, 92% of native invocations read tainted data within the first 10% of executed MAMBO blocks following a trap. The few executions near 100% are due to a single native method.

We also present a histogram in Figure 5b 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.

5.4 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 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, multiplying by 4V provides 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 6 shows the energy results for our Instagram experiments. We compare stock Android, SandTrap with tainted microphone data, SandTrap with tainted camera data, and SandTrap under continuous DIFT. As expected, SandTrap's energy usage increases with increased DIFT activity. Stock Android consumes a median of 181 Joules while SandTrap with tainted microphone data uses approximately 200 Joules. Recall that our Instagram workload does not process microphone data. SandTrap with tainted camera data and SandTrap under continuous DIFT both consume approximately 250 Joules. Note that in all experiments, SandTrap using memory domains consistently consumed 10 Joules fewer than the two-page-tables implementation.

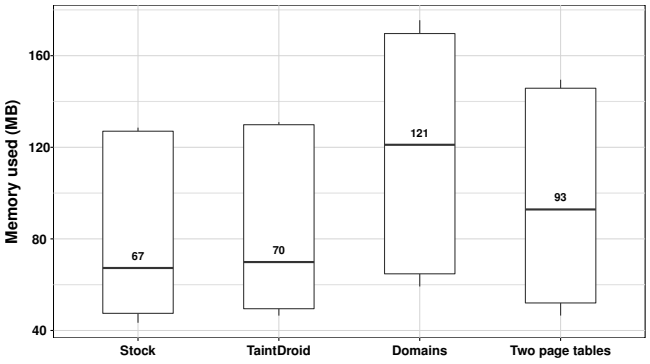


Figure 7: SandTrap memory consumption using Instagram.

5.5 Memory domains vs two page-tables

In the experiments above, SandTrap with memory domains outperforms SandTrap with two page-tables for several reasons. First, under domains, for a thread to switch from tracked to untracked, the kernel only needs to update the DACR. Under two 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 must 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, under memory domains physical pages under the `TRACKED` domain cannot be mapped to multiple virtual pages. To see if this limitation leads to higher memory usage, we run our Instagram experiments with stock Android, TaintDroid, and SandTrap with tainted camera data, and collect memory usage information with `procrank`, a tool on Android that shows each app's unique set size (the amount of memory private to that app). Figure 7 illustrates the results.

We sample this information at each discrete point of our Instagram experiment (e.g., before taking a photo, before annotating the image, and before uploading it). Therefore, our results represent the steady state memory usage. The domains implementation of SandTrap consumes the most memory, using a median of 121MB of memory. This is because when a 1MB region of memory is placed in the `TRACKED` domain, SandTrap immediately allocates a new copy of all pages that are shared or mapped to the zero page. In contrast, SandTrap with two page-tables can accommodate more physical page sharing and uses only 93MB of memory. Note that both SandTrap implementations use more memory than either stock Android or TaintDroid. This is due to the additional label storage that the SandTrap implementations require.

6 RELATED WORK

Section 2.3 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 [35], CHERI [36], and proposals for unlimited watchpoints [19] 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 [6] and lightweight contexts (lwCs) [24] 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, but neither allows threads to run in parallel. Secure Memory Views (SMV) [21] is also similar to SandTrap's parallel permissions in that it uses multiple page tables to provide different memory protections to threads running in parallel. However, SMV does not allow a thread to switch permissions depending on its execution context, and unlike pthreads, SMV threads do not share an address space by default. Both of these restrictions are a poor fit for our setting.

ARMLock [39] and Shreds [8] 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 [12] used memory domains to improve fork performance by consolidating the use of physical pages for storage page tables.

Finally, Dune [5] 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.

7 CONCLUSION

In this paper we have presented the design and implementation of SandTrap. SandTrap performs multithreaded, on-demand DIFT of native third-party 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.

8 ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers and our shepherd Aruna Balasubramanian for their invaluable feedback.

REFERENCES

[1] 2015. Memory protection keys. https://lwn.net/Articles/643797/. (2015).
[2] 2018. Android Developers - Keeping Your App Responsive. https://developer.android.com/training/articles/perf-anr.html. (2018).
[3] Andrew Appel and Kai Li. 1991. Virtual Memory Primitives for User Programs. In *Proceedings of ASPLOS '91.*
[4] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damein Octeau, and Patrick McDaniel. 2014. FlowDroid: Precice Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of PLDI '14.*
[5] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of OSDI '12.*
[6] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting Applications into Reduced-privilege Compartments. In *Proceedings of NSDI '08.*
[7] Erik Bosman, Asia Slowinska, and Herbert Bos. 2011. Minemu: The World's Fastest Taint Tracker. In *Proceedings of RAID '11.*
[8] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. 2016. Shreds: Fine-Grained Execution Units with Private Memory. In *Proceedings of IEEE SP '16.*
[9] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of ISSTA '07.*
[10] Landon P. Cox, Peter Gilbert, Geoffrey Lawler, Valentin Pistol, Ali Razeen, Bi Wu, and Sai Cheemalapati. 2014. SpanDex: Secure Password Tracking for Android. In *Proceedings of USENIX Security '14.*

[11] David Devecsery, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2018. Optimistic Hybrid Analysis: Accelerating Dynamic Analysis Through Predicated Static Analysis. In *Proceedings of ASPLOS '18 (ASPLOS '18)*.
[12] Xiaowan Dong, Sandhya Dwarkadas, and Alan L. Cox. 2016. Shared Address Translation Revisited. In *Proceedings of EuroSys '16*.
[13] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. 2005. Labels and Event Processes in the Asbestos Operating System. In *Proceedings of SOSP '05*.
[14] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. 2011. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of NDSS '11*.
[15] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-Flow Tracking system for Realtime Privacy Monitoring on Smartphones. In *Proceedings of OSDI '10*.
[16] WesLee Frisby, Benjamin Moench, Benjamin Recht, and Thomas Ristenpart. 2012. Security Analysis of Smartphone Point-of-Sale Systems. In *Proceedings of WOOT '12*.
[17] Peter Gilbert, Jaeyong Jung, Kyungmin Lee, Henry Qin, Daniel Sharkey, Anmol Sheth, and Landon P. Cox. 2011. YouProve: Authenticity and Fidelity in Mobile Sensing . In *Proceedings of SenSys '11*.
[18] Cosmin Gorgovan, Amanieu d'Antras, and Mikel Luján. 2016. MAMBO: A Low-Overhead Dynamic Binary Modification Tool for ARM. *ACM Trans. Archit. Code Optim.* 13, 1, Article 14 (April 2016), 14:1–14:26 pages.
[19] Joseph L. Greathouse, Hongyi Xin, Yixin Luo, and Todd Austin. 2012. A Case for Unlimited Watchpoints. In *Proceedings of ASPLOS '12*.
[20] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. 2006. Practical Taint-Based Protection using Demand Emulation. In *Proceedings of EuroSys '06*.
[21] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathis Payer. 2016. Enforcing Lease Privilege Memory Views for Multithreaded Applications. In *Proceedings of CCS '16*.
[22] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. 2011. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *Proceedings of NDSS '11*.
[23] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. 2007. Information Flow Control for Standard OS Abstractions. In *Proceedings of SOSP '07*.
[24] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-weight Contexts: An OS Abstraction for Safety and Performance. In *Proceedings of OSDI '16*.
[25] James Newsome and Dawn Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of NDSS '05*.
[26] Chenxiong Qian, Xiapu Luo, Yuru Shao, and Alvin T.S. Chan. 2014. On Tracking Information Flows through JNI in Android Applications. In *Proceedings of DSN '14*.
[27] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. 2006. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *Proceedings of MICRO '06*.
[28] Andrew Quinn, David Devecsery, Peter M. Chen, and Jason Flinn. 2016. JetStream: Cluster-Scale Parallelization of Information Flow Queries. In *Proceedings of OSDI '16*.
[29] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. 2012. AppInsight: Mobile App Performance Monitoring in the Wild. In *Proceedings of OSDI '12*.
[30] Ali Razeen, Valentin Pistol, Alexander Meijer, and Landon P. Cox. 2016. Better Performance Through Thread-local Emulation. In *Proceedings of HotMobile '16*.
[31] Haichen Shen, Aruna Balasubramanian, Anthony LaMarca, and David Wetherall. 2015. Enhancing Mobile Apps To Use Sensor Hubs Without Programmer Effort. In *Proceedings of UbiComp '15*.
[32] Riley Spahn, Jonathan Bell, Michael Z. Lee, Sravan Bhamidipati, Roxana Geambasu, and Geil Kaiser. 2014. Pebbles: Fine-Grained Data Management Abstractions for Modern Operating Systems. In *Proceedings of OSDI '14*.
[33] Mingshen Sun, Tao Wei, and John C.S. Lui. 2016. TaintART: A Practical Multi-level Information-Flow Tracking System for Android RunTime. In *Proceedings of CCS '16*.
[34] Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda. 2012. Clean OS: Limiting Mobile Data Exposure with Idle Eviction. In *Proceedings of OSDI '12*.
[35] Emmett Witchel, Josh Cates, and Krste Asanović. 2002. Mondrian Memory Protection. In *Proceedings of ASPLOS '02*.
[36] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceedings of ISCA '14*.
[37] Lok Kwong Yan and Heng Yin. 2012. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proceedings of USENIX Security '12*.
[38] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. 2006. Making Information Flow Explicit in HiStar. In *Proceedings of OSDI '06*.
[39] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. 2014. ARMlock: Hardware-based Fault Isolation for ARM. In *Proceedings of CCS '14*.