# Practical Dynamic Information-Flow Tracking on Mobile Devices

by

Valentin Pistol

Department of Computer Science
Duke University

Date: ____________________________

Approved:

____________________________
Alvin R. Lebeck, Co-Supervisor

____________________________
Landon P. Cox, Co-Supervisor

____________________________
Daniel J. Sorin

____________________________
Jeffrey S. Chase

____________________________
Bruce M. Maggs

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
2014

## Abstract

# Practical Dynamic Information-Flow Tracking on Mobile Devices

by

Valentin Pistol

Department of Computer Science
Duke University

Date: ______________________
Approved:

______________________
Alvin R. Lebeck, Co-Supervisor

______________________
Landon P. Cox, Co-Supervisor

______________________
Daniel J. Sorin

______________________
Jeffrey S. Chase

______________________
Bruce M. Maggs

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
2014

# Abstract

Today's consumer mobile platforms such as Android and iOS manage large ecosystems of untrusted third-party applications. It is common for an application to request one or more types of sensitive data. Unfortunately, users have no insight into how their data is used. Given the sensitivity of the data accessible by these applications, it is paramount that mobile operating systems prevent apps from leaking it.

This dissertation shows that it is possible to improve the soundness of dynamic information-flow tracking on a mobile device without sacrificing precision, performance, or transparency. We extend the state of the art in dynamic information-flow tracking on Android and address two major limitations: quantifying implicit flow leaks in Dalvik bytecode and tracking explicit flows in native code. Our goal is to deliver seamless end-to-end taint tracking across Dalvik bytecode and native code.

We propose SpanDex, a system that quantifies implicit flow leaks in Dalvik bytecode for apps handling password data. SpanDex computes a bound of revealed tainted data by recording the control-flow dependencies and for each password character, keeps track of the possible set of values that have been inferred. We also propose TaintTrap, a taint tracking system for native code in third party apps. We explore native taint tracking performance bottlenecks and hardware acceleration techniques to improve instrumentation performance.

# Contents

# List of Tables

# List of Figures

# List of Listings

# List of Abbreviations

| | |
|---|---|
| API | Application Programming Interface. |
| App | Application. |
| CPI | Cycles Per Instruction. |
| CPU | Central Processing Unit. |
| DIFT | Dynamic Information-Flow Tracking. |
| FE | Full Emulation. |
| OS | Operating System. |
| PC | Program Counter. |
| SE | Selective Emulation. |
| SP | Stack Pointer. |
| Syscall | System call. |
| TAP | Taint Access Protection. |
| TLS | Thread-Local Storage. |
| VM | Virtual Machine. |

# Acknowledgements

My parents, for their unbounded love and motivation.

My brother, Costi, for always being there for me, his support, insight and role-model example.

My advisors, Alvin Lebeck and Landon Cox, for their invaluable guidance and mentorship throughout my Ph.D.

# Chapter 1

# Dynamic Information-Flow Tracking

## 1.1 Introduction

There is a clear need for privacy and security in computer systems. A modern example comes from widespread adoption and usage of popular mobile devices such as smartphones and tablets. Users store increasing amounts of sensitive information on their personal devices but also make use of applications (*apps*) that do not offer privacy guarantees. Growing reports of security breaches, confidential and personal information leaks are now common events.

Current mobile OS's such as iOS or Android, provide the user the option of granting or denying access to certain types of information (e.g. location, contacts, photos). This leaves the user deciding whether to trust an app will not leak their sensitive information. Once an app is granted permissions, the user has no guarantee the same permissions are not used in a way that either intentionally or unintentionally leaks sensitive information.

The underlying problem in todays mobile devices is that they make no guarantees in secure information-flow. As defined by the pioneering work of Denning [1] in the 70s, *secure information-flow* means no unauthorized flow of information is possible.

## 1.2 Motivation

Although there is a vast range of previous work on *dynamic information-flow tracking* (DIFT), their design goals and decisions were done in a very different context. One of the major distinctions is the environment: desktop versus smartphone. Unlike a desktop, a smartphone has different usage scenarios, more limited performance and is under strict battery constraints. We aim for a system that is continuously monitoring and protecting sensitive data and doing so without incurring an overhead that impairs users device interaction. In our view, the user does not trust the application developer with safe guarding sensitive data stored on the device from being leaked to external parties. Coupled with the vibrant app ecosystem on smartphones, we need a tracking system that can support unmodified app binaries whether from official or third party app stores.

To summarize, we argue for a practical smartphone DIFT system with the following properties:

- Full-system end-to-end dynamic information-flow tracking: support across OS, libraries and application code.
- Low performance overhead: minimal impact on noticeable user device performance.
- High tracking precision: fine-granularity accurate taint tracking (low false positives) and multiple taint markings.
- Online continuous monitoring and protection.
- No app developer burden (e.g. annotations).
- No application source code required.

In delivering a smartphone tracking system none of the prior works offer a practical system with all the desirable properties outlined above. Other work has extensively addressed some of the properties we seek but they are either unable to cover all the properties, have significant limitations or their performance overheads are not suitable for real-time usage on a smartphone. We think the time is ideal for providing such a desirable DIFT system.

In this work we focus specifically on the popular Android platform, mostly because it offers complete source code on which we can build on. Android provides a powerful mix of features: garbage collection (GC), virtual machine (VM), just-in-time (JIT) compilation and native code (ARM). All of these can have very different impact and requirements with respect to information-flow tracking, both in tracking granularity and overhead. Leveraging these features efficiently is one of our main goals.

We first discuss at a high level the motivation and challenges for the system properties we chose:

**Full-system end-to-end DIFT** Most prior work addresses application level information-flow. Moreover apps interact with the underlying system in many ways that could leak information. On Android, apps are composed of Dalvik bytecode running inside a VM and can optionally include native ARM code. In order to ensure strong privacy guarantees a DIFT system needs to be *end-to-end*, covering all interactions both between and within the OS and app layers. In the case of Android, we specifically need taint tracking across the OS, filesystem, network, Dalvik VM and across native code. To address this, in this paper we are extending the TaintDroid platform with precise native code taint tracking.

**Low performance overhead** If a DIFT system is to monitor sensitive information and provide privacy guarantees, it holds that the system needs to be continuously aware of any sensitive flows. One way current systems approach this is by instrumenting all machine instructions with additional instructions required to propagate and update taint. Generally these instructions occur regardless of any tainted data actually being propagated and they can incur very significant overheads. While for certain types of data flow or security analyses the performance overhead might not be a limiting factor, in the case of smartphones performance is critical. Not only is performance capability very limited on a smartphone compared to a desktop but the types of apps running on a device are generally highly interactive. This prohibits the use of systems that significantly increase overhead making device interaction extremely slow and unusable.

To reduce DIFT performance, some systems propose hardware extensions. In general, the hardware associates tags with each memory location and register, and propagates the tags in hardware, without any additional instrumentation overhead. These systems are generally limited to 1-bit tags with prohibitive overheads for multi-bit tags and have inflexible taint propagation rules. The drawback is that the requirement on highly specialized hardware limits wide-spread deployment and adoption, and the flexibility is generally constrained over a software approach. We believe a more interesting middle ground is in hybrid solutions that minimally extend or repurpose existing hardware together with software for high flexibility. In this work we favor the latter hybrid approach.

**Online continuous monitoring and protection** A key goal is to offer regular users a system that runs in real-time on their smartphone, watches and safeguards sensitive data at all times. Continuously running a typical heavy-weight DIFT system is not feasible especially on a severely resource constrained devices such as smartphones. We believe there is an opportunity to meet the goal of having continuous execution while still maintaing useable device performance by specifically tuning DIFT for Android with variable taint tracking granularity (e.g. file, parcel, object, field, variable and instruction levels).

**High tracking precision** The majority of DIFT systems offer only the ability to label or tag data with a single bit, representing trusted or untrusted data. While this is sufficient to detect some security attacks or leaks, it is not suitable for providing a rich policy set. Having more tag expressiveness by raising the number of bits per tag can accommodate for more user-focused policies: a user may consider sharing his coarse grain location with certain apps but not his fine-grain location, or may chose to explicitly deny apps the ability to send out his phone number or contacts without explicit approval, while allowing other information such as his IMEI that is generally used for advertisement tracking purposes. These policies could require around 5 bits (encoding 32 different types of tainted data). On the high end of tag expressiveness, using a full 32-bit tag can enable not just leak detection but precise flow tracking, determining what path a piece of data takes from its first use to its point of exit from the system. These types of analyses are called information-flow data tomography and can be very powerful.

**No developer burden** Some previously proposed systems offload to the developer the burden of annotating their programs. There are various approaches: rely on special information-flow programming languages and having to annotate sensitive data with labels, might be required to include and link against some special IFT aware libraries in their code to allow the platform to intercept sensitive flows, rely on a specially modified compiler to compute and verify program labels. While these approaches may be suitable for some applications, in our case where users can easily install applications from the many hundreds of thousands available in the app stores, it is infeasible to expect all developers will perform the extra effort. Moreover, it is likely the user does not trust the app developer with properly making such changes. The user is much more likely to trust the platform (Google in the case of Android or Apple for iOS) or a subset of the platform, responsible for enforcing privacy mechanisms. Thus in this work we are designing a system that does not require the user trust the developer or the developer perform extra effort to support a DIFT smartphone system.

**No application source code required** In a similar view to the previous property of no developer burden, we also make the important observation that even though some DIFT systems may not require developer burden, they still rely on the application source code to automatically instrument the original source code before running it. This is a major barrier as most applications offered on app stores, although free, do not offer their source code as well. Requiring source code is thus a significant limitation for a practical smartphone DIFT system. The user should never have to worry about these aspects and instead should expect the underlying system will function correctly and protect their data at all times regardless of what apps they install and from what sources. That being said, we note that although app source code is not available, we are free to modify or statically instrument the available app binary. For instance, we consider the one-time cost of binary rewriting acceptable. In the case of Android, the system already does some binary optimizations the first time an app is installed so it is possible we perform additional rewriting for taint tracking purposes.

### 1.2.1 Android

Android is a open source mobile device platform. The Android OS is Linux-based and it offers support for third party applications built on top of the platform. Apps are written in Java and compiled to Dalvik EXecutable (DEX) bytecode. Apps can also include optional C libraries and C code is compiled to ARM assembly. Each application runs in its own process which includes a Dalvik VM.

**Dalvik VM Interpreter** Dalvik VM process DEX bytecode in a register-based machine. The registers and stack are managed internally by the VM. Unlike other register-based machines like x86, Dalvik long-term storage is in the form of class fields as opposed to storing in memory locations. This property helps with taint propagation. For performance, Android also includes a just-in-time (JIT) compiler starting with Android 2.3.

**Native Methods** Android applications can be partially or fully written in C and compiled to native ARM assembly. This use case is common practice for Android platform libraries, 3rd party libraries such as OpenGL and Webkit, games or performance critical code. Until now, native code support for taint tracking on Android has been a limitation. There are two types of native methods: VM internal methods and JNI methods. The VM internal methods are used for interpreter structures and operation while the JNI implement the Java Native Interface standard [2].

Current Android tracking systems support only applications running inside the Dalvik VM. Taint tracking inside a VM has practical benefits: low performance overhead due to operation on Dalvik bytecode as opposed to lower level ARM instructions, isolation and metadata rich execution environment.

On the other hand, running native code escapes the VM and the benefits are lost at the gain of increased application performance. Malware applications can circumvent the VM taint tracking system by simply running native code.

**Binder IPC** The Android IPC system is Binder. A Binder IPC library runs in userspace while a kernel module manages communication between processes. The messaging unit in Binder is a *parcel*.

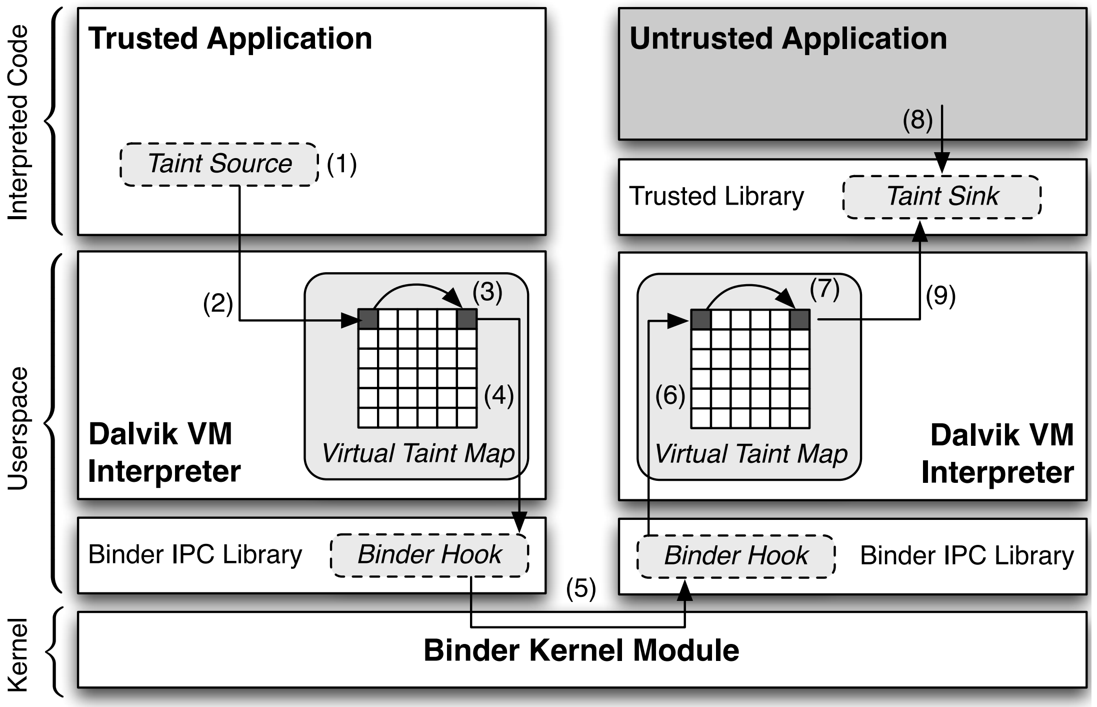

FIGURE 1.1: TaintDroid architecture within Android.

### 1.2.2 TaintDroid

TaintDroid is an enhancement built over Android that provides system-wide DIFT. The TaintDroid architecture within Android is shown in Figure 1.1. Taint tracking is performed with variable granularity allowing for low overhead. A single tag is 32-bit and stores 32 distinct 1-bit taint markings. Taint tags are associated with files, parcels, method local variables, method arguments, class fields and arrays. Files and parcels are associated a single tag. A limitation of TaintDroid is that taint tracking is not possible inside native methods. Currently TaintDroid only patches the return taint tag of a native method by considering the taint tags of all its input arguments. Figure 1.2 shows the modified stack format required for adding taint tag support for both interpreted targets and native targets.

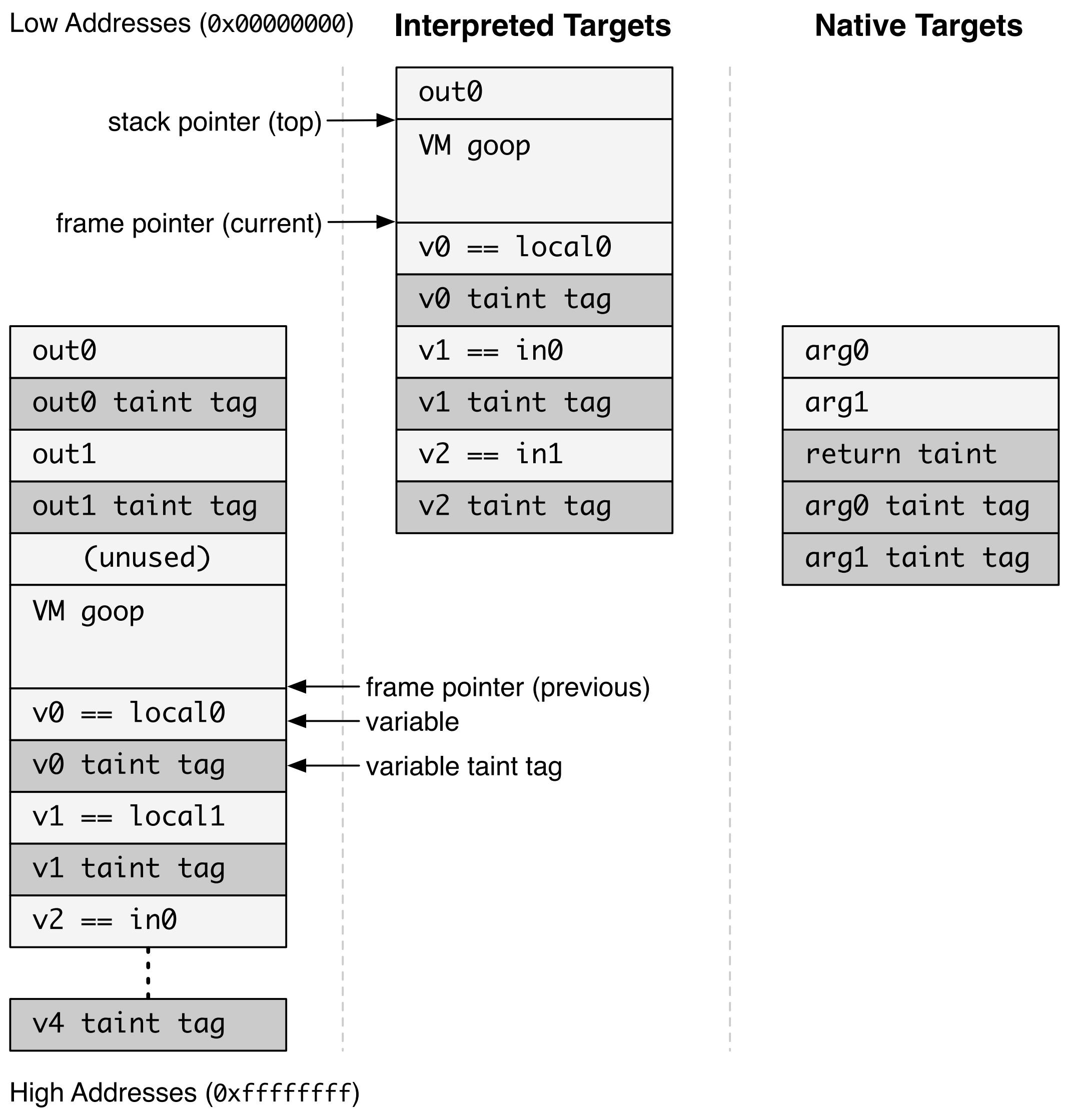

Figure 1.2: TaintDroid modified stack format.

# Chapter 2

# SpanDex: Secure Password Tracking for Android

## 2.1 Introduction

Today's consumer mobile platforms such as Android and iOS manage large ecosystems of untrusted third-party applications called "apps." Apps are often integrated with remote services such as Facebook and Twitter, and it is common for an app to request one or more passwords upon installation. Given the critical and ubiquitous role that passwords play in linking mobile apps to cloud-based platforms, it is paramount that mobile operating systems prevent apps from leaking users' passwords. Unfortunately, users have no insight into how their passwords are used, even as credential-stealing mobile apps grow in number and sophistication [3–5].

Taint tracking is an obvious starting point for securing passwords [6]. Under taint tracking, a monitor maintains a *label* for each *storage object*. As a process executes, the monitor dynamically updates objects' labels to indicate which parts of the system state hold secret information. Taint tracking has been extensively studied for many decades and has practical appeal because it can be transparently implemented below existing interfaces [6–9].

Most taint-tracking monitors handle only *explicit flows*, which directly transfer secret information from an operation's source operands to its destination operands. However, programs also contain *implicit flows*, which transfer secret information to objects via a program's control flow. Implicit flows are a long-standing problem [1] that, if left untracked, can dangerously understate which objects contain secret information. On the other hand, existing techniques for securely tracking implicit flows are prone to significantly *overstating* which objects contain secret information.

Consider secret-holding integer variable $s$ and pseudo-code **if** $s$ != 0 **then** $x := a$ **else** $y := b$ **done**. This code contains explicit flows from $a$ to $x$ and from $b$ to $y$ as well as implicit flows from $s$ to $x$ and $s$ to $y$. A secure monitor must account for the information that flows from $s$ to $x$ and $s$ to $y$, regardless of which branch the program takes: $y$'s value will depend on $s$ even when $s$ is non-zero, and $x$'s value will depend on $s$ even when $s$ is zero.

Existing approaches to tracking implicit flows apply static analysis to all untaken execution paths within the scope of a tainted conditional branch. The goal of this analysis is to identify all objects whose values are influenced by the condition. Strong security requires such analysis to be applied conservatively, which can lead to prohibitively high false-positive rates due to variable aliasing and context sensitivity [9, 10].

In this paper, we describe a set of extensions to Android's Dalvik virtual machine (VM) called SpanDex that provides strong security guarantees for third-party apps' handling of passwords. The key to our approach is focusing on the common access patterns and semantics of the data type we are trying to protect (i.e., passwords).

SpanDex handles implicit flows by borrowing techniques from symbolic execution to precisely quantify the amount of information a process' control flow reveals about a secret. Underlying this approach is the observation that as long as implicit flows transfer a safe amount of information about a secret, the monitor need not worry about where this information is stored. For example, mobile apps commonly branch on a user's password to check that it contains a valid mix of characters. As long as the implicit flows caused by these operations reveal only that the password is well formatted, the monitor does not need to update any object labels to indicate which variables' values depend on this information.

To quantify implicit flows at runtime without sacrificing performance, SpanDex executes untrusted code in a data-flow defined sandbox. The key property of the sandbox is that it uses data-flow information to restrict how untrusted code operates on secret data. In particular, SpanDex is the first system to use constraint-satisfaction problems (CSPs) at runtime to naturally prevent programs from certain classes of behavior. For example, SpanDex does not allow untrusted code to encrypt secret data using its own cryptographic implementations. Instead, SpanDex's sandbox forces apps that require cryptography to call into a trusted library.

*SpanDex does not "solve" the general problem of implicit flows.* If the amount of secret information revealed through a process' control flow exceeds a safe threshold, then a monitor must either fall back on conservative static analysis for updating individual labels or simply assume that all subsequent process outputs reveal confidential information. However, we believe that the techniques underlying SpanDex may be applicable to important data types besides passwords, including credit card numbers and social security numbers. Experiments with a prototype implementation demonstrate that SpanDex is a practical approach to securing passwords. Our experiments show that SpanDex generates far fewer false alarms than the current state of the art, protects user passwords from a strong attacker, and is efficient.

This paper makes the following contributions:

- SpanDex is the first runtime to securely track password data on unmodified apps at runtime without overtainting or poor performance.

- SpanDex is the first runtime to use online CSP-solving to force untrusted code to invoke trusted libraries when performing certain classes of computation on secret data.

- Experiments with a SpanDex prototype show that it imposes negligible performance overhead, and a study of 50 popular, non-malicious unmodified Android apps found that all but eight executed normally.

The rest of this paper is organized as follows: Section 2.2 describes background information and our motivation, Section 2.3 provides an overview of SpanDex's design, Section 2.4 describes SpanDex's design in detail, Section 2.5 describes our SpanDex prototype, Section 2.6 describes our evaluation, and Section 2.7 provides our conclusions.

## 2.2 Background and Motivation

Under dynamic information-flow tracking (i.e., taint tracking), a monitor maintains a *label* for each *storage object* capable of holding secret information. A label indicates what kind of secret information its associated object contains. Labels are typically represented as an array of one-bit *tags*. Each tag is associated with a different source of secret data. A tag is set if its object's value depends on data from the tag's associated source. *Operations* change objects' state by transferring information from one set of objects to another. Monitors *propagate tags* by interposing on operations that could transfer secret information, and then updating objects' labels to reflect any data dependencies caused by an operation. We say that information derived from a secret is *safe* if it reveals so little about the original secret that releasing the information poses no threat. However, if information is *unsafe*, then it should only be released to a trusted entity.

### 2.2.1 Related Work: Soundness, Precision, and Efficiency

The three most important considerations for taint tracking are soundness, precision, and efficiency. Tracking is *sound* if it can identify all process outputs that contain an unsafe amount of secret information. Soundness is necessary for security guarantees, such as preventing unauthorized accesses of secret information. Tracking is *precise* if it can identify how much secret information a process output contains. Precision can be tuned along two dimensions: better *storage precision* associates labels with finer-grained objects, and better *tag precision* associates finer-grained data sources with each tag.

Imprecise tracking leads to *overtainting*, in which safe outputs are treated as if they are unsafe. A common way to compensate for imprecise tracking is to require users or developers to *declassify* tainted outputs by explicitly clearing objects' tags.

Tracking is *efficient* if propagating tags slows operations by a reasonable amount. The relationship between efficiency and precision is straightforward: increasing storage precision causes a monitor to propagate tags more frequently because it must interpose on lower-level operations; increasing tag precision causes a monitor to do more work each time it propagates tags. Finding a suitable balance of soundness, precision, and efficiency is challenging, and prior work has investigated a variety of points in the design space.

One approach to information-flow tracking is to use static analysis in combination with a secrecy-aware type system and programmer-defined declassifiers to prevent illegal flows [11]. This approach is sound, precise, and efficient but is not compatible with legacy apps. Integrating secrecy annotations and declassifiers into apps and platform libraries requires a non-trivial re-engineering effort by developers and platform maintainers.

An alternative way to ensure soundness is to propagate tags on high-level operations that generate only *explicit flows.* An explicit flow occurs when an operation directly transfers information from from a set of well-defined source objects to a set of well-defined destination objects [1]. For example, process-level monitors such as Asbestos [12], Flume [13], and HiStar [14] maintain labels for each address space and kernel-managed communication channel (e.g., file or socket), and propagate tags for each authorized invocation of the system API.

Such process-grained tracking is sound and efficient, but operations defined by a system API commonly manipulate fine-grained objects, such as byte ranges of memory. The mismatch between the granularity of labeled objects and operation arguments leads to imprecision. For example, once a process-grained monitor sets a tag for an address space's label, it conservatively assumes that any subsequent operation that copies data out of the address space is unsafe, even if the operation discloses no secret information.

As with language-based flow monitors, process-grained monitors must rely on trusted declassifiers to compensate for this imprecision. These declassifiers proxy all inter-object information transfers and are authorized to clear tags from labels under their control. However, because declassifiers make decisions with limited context, they can be difficult to write and require developers to modify existing apps.

Other monitoring schemes have improved precision by associating labels with finer-grained objects such as individual bytes of memory [7, 8]. While tracking at too fine a granularity leads to prohibitively poor performance [7, 8] (e.g., 10x to 30x slowdown), propagating tags for individual variables within a high-level language runtime is efficient [6]. The primary challenge for such fine-grained tracking is balancing soundness and precision in the presence of *implicit flows.*

As before, consider secret-holding variable $s$ and pseudo-code **if** $s$ != 0 **then** $x := a$ **else** $y := b$ **done**. Borrowing terminology from [15], we say that all operations between **then** and **done** represent the *enclosed region* of the conditional branch. Thus, the enclosed region contains explicit flows from $a$ to $x$ and from $b$ to $y$. Operations like conditional branches induce implicit flows by transferring information from the objects used to evaluate a condition to any object whose value is influenced by an execution path through the enclosed region. We refer to the set of influenced objects as the *enclosed set*. The enclosed set includes all objects that are modified along the taken execution path as well as all objects that *might have been modified* along any untaken paths. To ensure soundness, a monitor must propagate $s$'s tags to all objects in the enclosed set.

Propagating tags to members of the enclosed set can lead to overtainting in two ways. First, because a conditional branch does not specify its enclosed set, the membership must be computed through a combination of static and dynamic analysis [8, 15]. In our example, a simple static analysis of the program's control-flow graph could identify the complete enclosed set consisting of $x$ and $y$. However, strong soundness guarantees require an overly conservative analysis of far more complex untaken paths containing context-sensitive operations and aliased variables. This can overstate which objects' values are actually influenced by a branch. Less conservative tag propagation creates opportunities for malicious code to leak secret information.

Second and more important, the amount of information transferred through a process' control flow is often very low. These information-poor flows expose the problem with tag imprecision. In particular, conventional monitors can only account for an implicit flow by propagating single-bit tags from the branch condition to members of the enclosed set. And yet members of the enclosed set can only reflect as much new information as the branch condition reveals. When the condition reveals very little information (e.g., $s$ != 0), a single-bit tag cannot be used to differentiate between an object whose value is weakly dependent on secret information and one whose value encodes the entire secret. Thus, when an execution's control flow transfers very little information, propagating tags to members of the enclosed set significantly overstates how much secret information the branch transfers to the rest of the program state.

Prior work on DTA++[9] and Flowcheck [15] have articulated similar insights about the causes of overtainting. DTA++ propagates tags to an enclosed set only if an execution's control flow reveals the entire secret (i.e., the execution path is injective with respect to a secret input). However, DTA++ relies on offline symbolic execution of several representative inputs to select which branches should propagate tags to their enclosed sets. Offline symbolic execution provides limited code coverage for moderately complex programs and is unlikely to deter actively malicious programs.

Flowcheck focuses on the imprecision of single-bit taint tags and precisely quantifies the total amount of secret information an execution reveals (as measured in bits). However, Flowcheck imposes significant performance penalties and must compute the enclosed set (often with assistance from the programmer) to quantify the channel capacity of enclosed regions.

To summarize, we are unaware of any prior work on information-flow tracking that provides a combination of soundness, precision, and efficiency that would be suitable for tracking passwords on today's mobile platforms.

### 2.2.2 Android-app Study

To test our hypothesis that conventional handling of implicit flows leads to overtainting and false alarms, we created a modified version of TaintDroid [6] called TaintDroid++ that supports limited implicit-flow tracking. TaintDroid and TaintDroid++ track explicit flows the same way. Each variable in a Dalvik executable is assigned a label consisting of multiple tags, and tags are propagated according to a standard tag-propagation logic.

The primary difference between the two monitors is that TaintDroid ignores implicit flows and TaintDroid++ does not. First, for a Dalvik executable, TaintDroid++ constructs a control-flow graph and identifies the immediate post-dominator (ipd) for each control-flow operation. It then uses smali [16] to insert custom Dalvik instructions that annotate (1) each ipd with a unique identifier, and (2) each control-flow operation with the identifier of its ipd. Like Dytan [8], TaintDroid++ does not propagate tags to objects that might have been updated along untaken execution paths.

Using these two execution environments, we ran four popular Android apps that require a user to enter a password: the official apps for LinkedIn, Twitter, Tumblr, and Instagram. Both systems tagged password data as it was input but before it was returned to an app. We then manually exercised each app's functionality and monitored its network and file outputs for tainted data.

Figure 2.1 shows the number and type of tainted outputs we observed for apps running under TaintDroid and TaintDroid++. For each tainted output, we manually inspected the content to determine whether it contained password data or not. Each tainted output under TaintDroid appeared to be an authentication message that clearly contained a password. TaintDroid++ also tainted these outputs, but generated many more tainted network and file writes. We were unable to detect any password information in these extra tainted outputs, and regard them as evidence of overtainting.

Overtainting is only a problem if incorrectly tainted data is copied to an inappropriate sink. Thus, a false positive occurs when an app copies data that is safe but tainted to an inappropriate sink. Apps authenticate using the OAuth protocol and should not store a local copy of a password once they receive an OAuth token from a server. Thus, each tainted file write generated under TaintDroid++ is a false positive.

For network writes, we also consider whether the password data was sent over

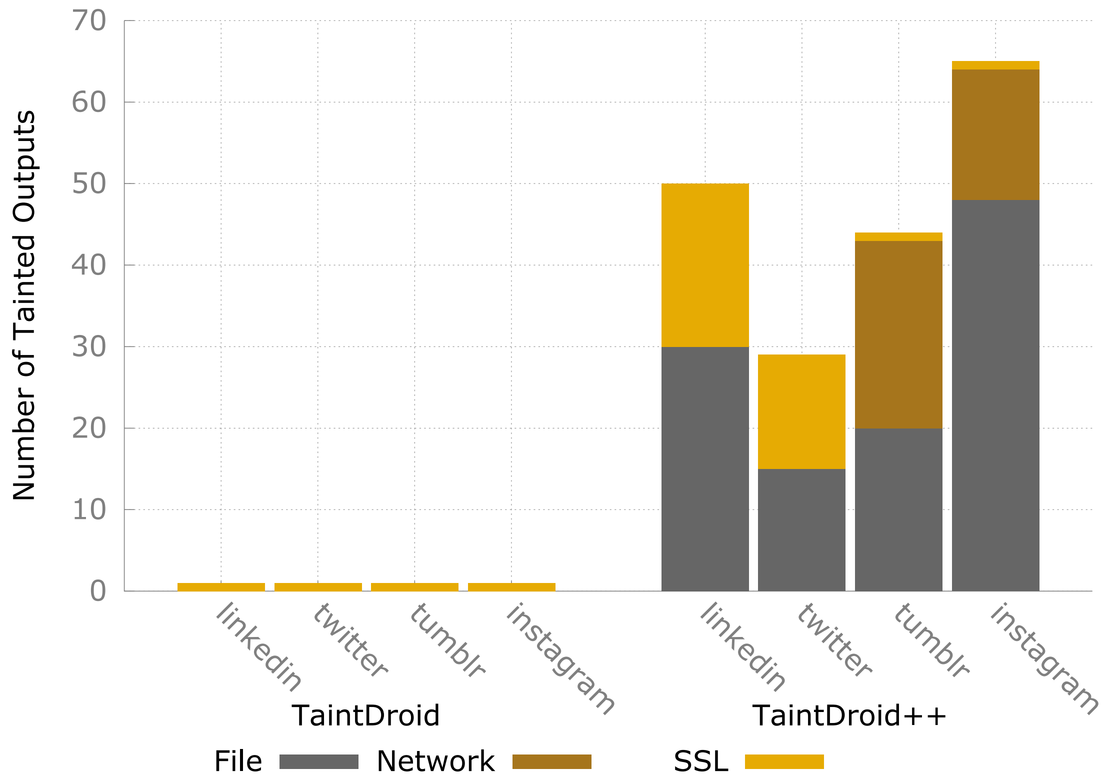

FIGURE 2.1: Tainted outputs for apps running under TaintDroid and TaintDroid++.

an encrypted connection (i.e., over SSL) and the IP address of the remote server. Both Tumblr and Instagram under TaintDroid++ generated unencrypted tainted network writes. None of these writes were tainted under TaintDroid. Furthermore, TaintDroid only taints outputs to appropriate servers, but under TaintDroid++ several overtainted outputs were sent to third-parties such as the cloudfront.net CDN and flurry.com analytics servers. These results are consistent with previous work on overtainting [17, 18], and confirm that securing users' passwords requires a better balance of soundness and precision.

## 2.3 System Overview

This section provides an overview of SpanDex, including the principles and attacker model that inform its design.

### 2.3.1 Principles

SpanDex's primary goal is to soundly and precisely track how information about a password circulates through a mobile app. For example, if an app requests a Facebook password, then SpanDex should raise an alert only if the app tries to send an unsafe amount of information about the password to a non-Facebook server. Preventing leaks also requires a way for users to securely enter and categorize their passwords, and to address these issues we rely on secure password-entry systems such as ScreenPass [19]. SpanDex is focused on tracking information *after* a password has been securely input and handed over to an untrusted app. The following design principles guided our work.

**Monitor explicit and implicit flows differently.** In practice, explicit and implicit flows affect a program's state in very different ways. Operations on secret data that trigger explicit flows, transfer a relatively large amount of secret information to a small number of objects. The inverse is true of control-flow operations that depend on secret data. These operations often transfer very little secret information to members of a large enclosed set. These observations led us to apply different mechanisms to tracking explicit and implicit flows.

First, SpanDex uses conventional taint tracking to monitor explicit flows. SpanDex is integrated with TaintDroid and Android's Dalvik VM, and maintains a label for each program variable. Each label logically consists of a single-bit tag indicating whether the variable contains an unsafe amount of information about a character within a user's password. Because explicit flows transfer a relatively large amount of information between objects, when an object's tag is set, SpanDex assumes that the variable contains an unsafe amount of secret information.

Second, when SpanDex encounters a branch with a tainted condition, it does not immediately propagate tags to objects in the enclosed set. Rather, SpanDex first updates an upper bound on the total amount of secret information the execution's control flow has revealed to that point. This upper bound precisely captures the maximum amount of secret information that an attacker could encode in untagged objects. As long as the total amount of secret information transferred through implicit flows is safe, SpanDex can ignore where that information is stored.

Like DTA++, SpanDex borrows techniques from symbolic execution to quantify the amount of information revealed through implicit flows. In particular, SpanDex integrates operation logging with tag propagation to record the chain of operations leading from a tainted variable's current state back to the original secret input. When SpanDex encounters a tainted conditional branch, it updates its information bounds by using these records to solve a constraint-satisfaction problem (CSP). The CSP solution identifies a set of secret inputs that could have led to the observed execution path. This set precisely captures the amount of information transferred through implicit flows.

The drawback of applying these techniques at runtime is the potential for poor performance. A monitor can efficiently record operations on tainted data at runtime, but solving a CSP when encountering a tainted branch could be disastrous. In the worst case, trying to solve a CSP could cause a non-malicious app to halt. For example, passwords must be encrypted before they are sent over the network, but it is infeasible to compute the set of all plaintext inputs that could have generated an encrypted output. Balancing the need to track implicit flows while preventing common primitives such as cryptography from slowing, or even halting, non-malicious apps led to our second design principle.

**If commonly used functionality makes tracking difficult, force apps to use a trusted implementation.** Mobile apps typically receive a password, perform sanity checks on the characters, encode the password as an http-request string, encrypt the http-request, and forward the encrypted string to a server. The code used to transform password data from one representation to another (e.g., encoding a character array as an http-request string and then encrypting the string) is problematic because it uses a number of operations that make quantifying implicit flows prohibitively slow or even impossible. This code includes a large number of bit-wise and array-indexing operations interleaved with tainted conditional branches. If SpanDex tracked implicit flows within this code as we have described thus far, non-malicious apps would become unusable.

Fortunately, it is exceedingly rare for apps to implement this functionality themselves. Instead, apps rely on platform libraries for common transformations, such as character encoding and cryptography. On Android this library code is small in size, easy to understand, and protected by the Java type system.

Tracking explicit flows remains the same for trusted libraries as for untrusted app code. However, within a trusted library, SpanDex does not solve CSPs when encountering a tainted branch and may directly update the information bound of a secret before exiting. This approach is sound for library code whose state is strongly encapsulated and whose semantics are well understood.

For example, encrypting a tainted string involves a sequence of calls into a crypto library for initializing the algorithm's state, updating that state, and retrieving the final encrypted result. Ignoring tainted conditional branches within this code is sound for two reasons. First, tracking explicit flows within the library ensures that any intermediate outputs as well as the final output are properly tagged. Second, external code can only access library state through the narrow interface defined by the library API; there is no way for untrusted code to infer properties about the plaintext except those that the library explicitly exposes through its interface or by branching on the plaintext data itself. SpanDex tracks both cases.

The protection boundary separating untrusted code from trusted library code has two novel properties. First, the boundary is defined by both data flow and control flow. An app is allowed to use a custom cryptographic implementation on untainted data, but must use the trusted crypto library to encrypt tainted data. Second, the boundary is enforced by the aggregate complexity of the operations performed rather than by hardware or a conventional software guard. If an app attempts to encrypt password data using a custom implementation *or* branches on encrypted data returned by the trusted library, it will be forced to solve an intractable CSP and halt.

Thus, the key property of a SpanDex's sandbox is that it restricts the classes of computation that untrusted code may directly perform on secret data. Instead, an app must yield control to the trusted platform so that these computations can be performed on its behalf.

Given an execution environment that can efficiently quantify the amount of secret information transferred through implicit flows, SpanDex's final challenge is determining whether the quantified amount is safe to release. This challenge led to our final design principle.

**Use properties of a secret's data type to set release policies.** Like SpanDex, DTA++ requires a threshold on the amount of information revealed through implicit flows. DTA++ applies a strict policy to determine when to propagate tags by doing so only when the control flow is injective. That is, DTA++ propagates tags when a single secret value could have led to a particular execution path.

Though simple, this policy is inappropriate for SpanDex. Revealing an entire secret value via implicit flows is clearly unsafe, but revealing partial information about a password may be too. For example, using carefully crafted branches, malware could cause significant harm by narrowing every character of a password to two possible values. However, as we have seen, treating all implicit flows as unsafe leads to prohibitive overtainting. SpanDex's challenge is to support practical release policies that sit between these two extremes.

SpanDex benefits from its focus on passwords. Passwords have a well-defined representation and fairly well understood attacker model. For example, it is reasonable to assume that an attacker knows that a password consists of a sequence of human-readable characters (i.e., ASCII characters 32 through 126), many of which are likely to be alphanumeric. An attacker gains no new information from observing the control flow of a process if the flow reveals that each character is within the expected range of values. We investigate what apps' control flows reveal in Section 2.6.

### 2.3.2 Trust and Attacker Model

SpanDex is implemented below the Dalvik VM interface (i.e., the Dex bytecode ISA), and the protections provided by this VM provide the foundation for SpanDex's trust model. Most Android app logic is written in Java and compiled into Dex bytecodes, which run in an isolated Dalvik VM instance. SpanDex cannot protect passwords from an app that executes third-party native code *while there is password data in its address space.* Thus, objects tainted with password data must be cleared before an app is allowed to execute its own native code. In addition, once a process invokes third-party native code, it may not receive password data. SpanDex must rely on the kernel to maintain information about which processes have invoked third-party native code. Finally, apps may not write tainted data to persistent storage or send it to another app via IPC.

SpanDex is focused on securely tracking how password data flows within an app.

Attacks on other aspects of password handling are outside the scope of our design. First, we assume that users can securely enter their password before it is given to an app, and that users will tag a password with its associated domain. A secure, unspoofable user interface, such as the one provided by ScreenPass [19], can provide such guarantees. Special purpose hardware, such as Apple's Touch ID fingerprint sensor and secure enclave [20], could also provide this guarantee.

Second, SpanDex can help ensure that password data is shared only with servers within the domain specified by the user, but provides no guarantees once it leaves a device. For example, SpanDex cannot prevent an attacker from sending a user's Facebook password as a message to a Facebook account controlled by the attacker. Preventing such cases requires cooperation between SpanDex and the remote server. SpanDex could notify the service when a message contains password data, and the service could determine whether such messages should contain password data.

We assume that an attacker completely controls one or more apps that a user has installed, and that the attacker is also in control of one or more remote servers. The attacker's servers can communicate with the attacker's apps, but the servers reside in a different domain than the one the user associates with her password. The attacker can make calls into the platform libraries and manipulate its apps' data and control flows to send information about passwords to its remote servers.

Based on the large-scale leakage of large password lists from major services, such as Gawker [21] and Sony Playstation [22], we assume that an attacker has access to a large list of unique passwords, and that the user's password is on the list. However, we assume that the attacker does not know which usernames are associated with each entry in the list (though it does know the user's username).

Thus, our attacker's goal is to de-anonymize the user within its password list using information gathered from its apps. The attacker can send its servers as much untainted data describing a user's password as SpanDex's release policies allow (i.e., the password length as well as a range of possible values for each password character). In the worst case, the attacker will eliminate all but one of the passwords in its list. On the other hand, if the app provides no new information, then the user's password could be any of the ones in the list.

Once the attacker has computed the set of possible passwords for a username, it can only identify the correct username-password combination through online querying. For example, if an attacker infers that Bob's Facebook password is one of ten possibilities, then the attacker needs at most ten tries to login to Facebook as Bob.

The attacker may also have extra information about the usage distribution of passwords in its database. For example, the attacker may know that one password is used by twice as many users as another. While information from the app can help the attacker narrow a user's password to a smaller set of possibilities, the usage distribution allows the attacker to prioritize its login attempts to reduce the expected number of attempts before a successful login. We return to this issue in Section 2.6.

## 2.4 SpanDex Design

As with conventional taint tracking, SpanDex updates objects' labels on each operation that generates an explicit flow. If the monitor encounters a control-flow operation with a tainted condition, it does not update the labels of objects in the enclosed set. Instead, the monitor updates an upper bound on the amount of information the execution's control flow has revealed about the secret input.

SpanDex represents this bound as a *possibility set (p-set).* SpanDex maintains a p-set for each password character an app receives. P-sets logically contain the possible values of a character revealed by a process' control flow. Each time the app's control flow changes as a result of tainted objects, SpanDex attempts to remove values from the secret's p-set.

### 2.4.1 Operation dag

In order to narrow a p-set, SpanDex must understand the data flow from the original secret values to a tainted condition. We capture these dependencies in an *operation dag (op-dag).* This directed acyclic graph provides a record of all taint-propagating operations that influenced a tainted object's value as well as the order in which the operations occurred.

SpanDex reuses TaintDroid's label-storage strategy, and stores each 32-bit label adjacent to its object's value. However, whereas each bit in a TaintDroid label represents a different category of sensitive data (e.g., location or IMEI), SpanDex labels are pointers to nodes in the op-dag. If an object's label is null, then it is untainted. If an object's label is non-null, then its value depends on secret data.

Label storage in SpanDex most significantly differs from TaintDroid for arrays. In TaintDroid, each array is assigned a single label for all entries. If any array element becomes tainted, then the entire array is treated as tainted. This approach is inappropriate for SpanDex because we want to track individual password characters. Thus, SpanDex maintains per-entry labels. However, the reason that TaintDroid maintains a single label for each array is storage overhead. Byte and character arrays account for a large percentage of an app's memory usage, and assigning a 32-bit label for each byte-array entry could lead to a minimum fourfold increase in memory overhead for array labels.

To avoid this overhead, SpanDex allocates labels for arrays only after they contain tainted data. Each array is initially allocated a single label. If the array is untainted, then its label points to null. If the array contains tainted data, then its label points to a separate label array, with one label for each array entry. As with local-variable and object-field labels, array-element labels point to nodes in the op-dag. Since very few arrays contain password data, the overhead of maintaining per-entry labels is low overall.

The roots of the op-dag are special nodes that contain the original value of each secret (i.e., each password character), a pointer to the secret's p-set, and domain information. A p-set is represented as a doubly-linked list of value ranges. Each entry in the list contains a pointer to the previous and next entries, as well as a minimum and maximum value. Ranges are inclusive, and the union of the ranges specifies the set of possible secret values revealed by an app's control flow. SpanDex initializes p-sets to the range $[32, 126]$ to represent all printable ASCII characters. A secret's domain can be specified by the user through a special software keyboard [19].

Each tainted object version has an associated non-root node that records the operation that created the version, including its source operands. Source operands can be stored as concrete values (when operands are untainted) or as pointers to other nodes in the op-dag (when operands are tainted).

A node can point to more than one node, and there may be multiple paths from a node to one or more roots. The more complex the paths from a node to the op-dag roots are, the more complex updating p-sets becomes.

## 2.4.2 Example Execution

If a tainted variable influences an app's control flow (e.g., via a conditional branch), then SpanDex traverses the op-dag from the node pointed to by the object's label toward the roots. To demonstrate how SpanDex maintains and uses op-dags and p-sets, consider the simple snippet of pseduo-code below. Figure 2.2 shows the resulting op-dag and p-set.

```
0000: mov v1, v0          // v0, v1 label=ROOT
0002: add v2, v1, 3       // v2's label=N1
0004: add v2, v2, 2       // v2's label=N2
0006: sub v3, 6, v2       // v3's label=N3
0008: add v2, v2, 7       // v2's label=N4
```

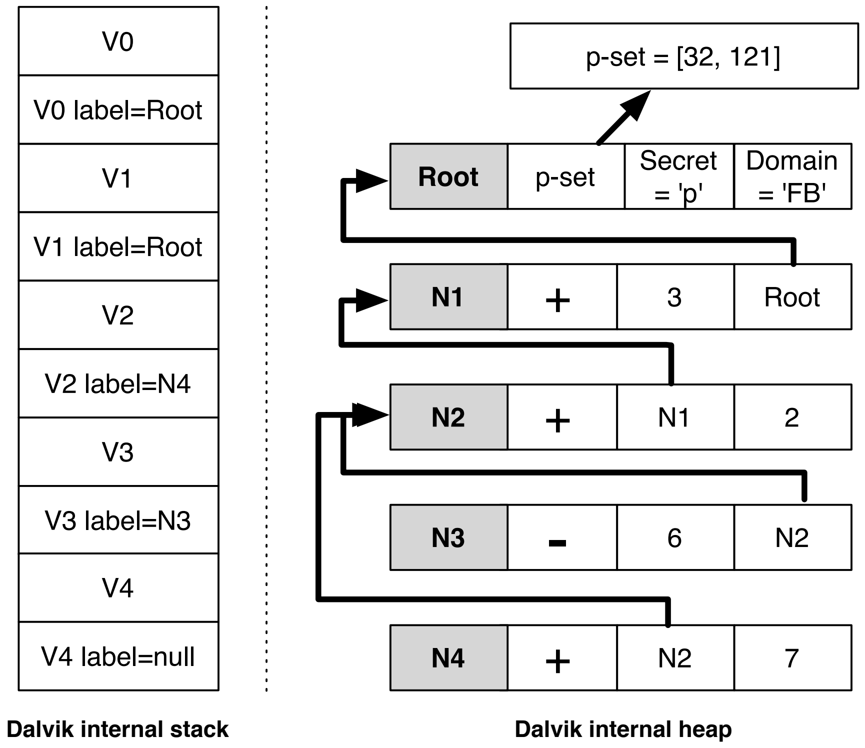

FIGURE 2.2: Simple op-dag and p-set example.

```
000a: const/16 v4, 122   // v4's label=0

000c: if-le v3, v4, 0016

000e: ...
```

The first character of the password is ‘p’, or numeric value 112, and is stored in register v0. The password’s domain is Facebook. v0’s label points to the Root node for the secret character. v0 is then copied into v1, whose label must also point to Root. The sum of v1 and 3 is then stored in v2, whose label then points to new node, N1. N1 contains the addition operation, the 3 operand, and points to Root. The next line adds 2 to v2. This creates a new version of v2, which is recorded in N2. N2 contains the 2 operand and points to the node for the previous version of v2, node N1. The remaining arithmetic operations proceed similarly. Finally, the code loads the constant value of 122 into v4 for an upcoming conditional branch. v4’s label is null, since it is not tainted.

When the code reaches the conditional branch, v3 is less than or equal to v4, since v3's value is 111, and v4's value is 122. Because v3's label is non-null, SpanDex uses the op-dag node in v3's label (N3) to update the p-set.

Updating the p-set is equivalent to solving a CSP to determine which secret values could have led to the control-flow change. In our example, updating the p-set is easy. SpanDex solves the inequality $v0 + 6 - 2 - 3 \leqslant 122$, leading to $v0 \leqslant 121$. Thus, the control flow reveals that the first character of the user's password is within the range of $[32, 121]$. SpanDex updates the p-set to reflect this before resuming execution. Figure 2.2 shows the state of the op-dag and p-set at this point.

This simple example demonstrates some of the challenges and nuances of SpanDex's approach. First, each node in the op-dag represents a version of a tainted variable. N3 points to the version of v2 used to update v3, so that when SpanDex reaches the conditional branch, it can retrieve the sequence of operations that led to v3's current value.

Second, reversible operations such as addition and subtraction make updating p-sets straightforward. Unfortunately, Dalvik supports a number of instructions that are much trickier to handle. For example, Dalvik supports instructions for operating on Java Object references and arrays that behave very differently than simple arithmetic operations. Even some classes of arithmetic operations, such as bit-wise operators and division, can make solving a CSP non-trivial.

Third, there was a single path from N3 to Root in our example. If N3 had forked due to multiple tainted operands, or had led to multiple root nodes due to mixing secret characters, solving the CSP would have been far more complex. Compression and cryptography often mix information from multiple characters, which creates a complex nest of paths from nodes to the op-dag roots.

Fortunately, among the popular non-malicious apps that we have studied, difficult-to-handle operations occur only in platform code such as the Android cryptography library. Furthermore, it is rare to find apps that branch on the results of these operations outside of platform code. Thus, as long as SpanDex can ensure that all outputs from these libraries are explicit and tainted, then we can ignore implicit flows within them (and thus, avoid CSP solving).

This approach is intuitive. First, outside of simple sanity checking on a password, there is little reason for an app to operate on password data itself. Second, libraries such as a crypto library are designed to suppress implicit flows. Observing an encrypted output or a cryptographic hash should not reveal anything about the plaintext input. Third, there is no obvious reason why app code should branch on either encrypted data or a cryptographic hash. Apps simply use the platform libraries to encode these outputs as strings and send them to a server.

There are many difficult operations that we have not observed in either app code or library code. Our general approach to these operations is to propagate taint to the results of these operations, but to fault if they cause the control flow to change. For example, an app may use bit-wise operations to encode a character, but branching on the encoded result is not allowed. This is secure and does not disrupt non-malicious apps. In the next section, we describe how SpanDex treats each class of Dex bytecodes in greater detail.

### 2.4.3 Dex Bytecodes

In this section, we describe how SpanDex handles each of the following classes of bytecodes: type-conversion operations, object operations, control-flow operations, arithmetic operations, and array operations.

**Type conversions.** Dalvik supports the following data types: boolean, byte, char, short, int, long, float, double, and Object reference, as well as arrays of each of these types. P-set ranges are represented internally as pairs of floats. Solving CSPs involving conversions to alternate representations is supported as long as the type is a native and numeric.

**Object Conversions.** Dex provides a number of instructions for converting between data types, but conversions can also occur through Object-method invocations and arrays. For example, an app could index into an Object array with a tainted character, where the fields of each Object encodes its position in the array. The returned Object reference would be tainted, and would identify the character used to index into the array. The return value of any method used to access a field of the tainted Object would also be tainted. However, SpanDex would have to understand the internal semantics of the Object in order to solve a CSP involving the tainted returned value. Thus, branching on data derived from a tainted Object reference is not allowed.

**Control-flow.** A Dalvik program's control flow can change as a result of secret data in many ways. Conditional branch operations such as `if-eq` are the most straightforward, and SpanDex handles these as described in Section 2.4.2.

Dalvik also supports two case switching operations: `packed-switch` and `sparse--switch`. Both instructions take an index and a reference to a jump table as arguments. The difference between the instructions is the format of the jump table and how it is used. The table for a `packed-switch` is a list of key-target pairs, in which the keys are consecutive integers. Dalvik first checks to see if the index is within the table's range of consecutive keys. If it is not, then the code does not branch and execution resumes at the instruction following the switch instruction. If it is in the table, then the code computes the new PC by adding the matching target to the current PC.

The table for a `sparse-switch` is also a list of key-target pairs, but the keys do not have to be consecutive integers (though they have to be sorted from low-to-high). To handle this instruction, the VM checks whether the index is greater than zero and less than or equal to the table size. It then uses the index to perform a binary search on the keys to find a match. If it finds a match, then it jumps to the instruction at the sum of the matching target and current PC.

Although more complex than conditional branches, handling these switch instructions is straightforward. If the code falls through the switch instruction, then the resulting implicit flow reveals that the index is not equal to any of the table keys. SpanDex can solve a CSP for each of the keys and update its p-sets accordingly. If the control-flow is diverted by the switch instruction, then the resulting implicit flow reveals that the index is equal to the matching table key. SpanDex can solve a CSP for this condition as well. In practice, most switch instructions are packed and the corresponding jump tables are small, which makes solving CSPs for these operations fast.

Finally, a program's control flow can be influenced by tainted data if an operation on tainted data causes an exception to be thrown. For example, an app could divide a number by a tainted variable with value of a zero, or it could use a tainted variable to index beyond the length of an array. SpanDex could compute a CSP for the information revealed by each of these conditions, e.g., that a tainted variable is equal to zero or that a tainted variable is greater than the length of an array. However, we have not seen this behavior in any of the apps we have studied. As a result, our current implementation simply stops the program when an instruction with a tainted operand causes an exception to be thrown.

**Arithmetic.** As we saw in Section 2.4.2, reversible arithmetic operations are straightforward to handle. Other arithmetic operations are not impossible to handle, but require a complex solver. For example, reversing multiplication and division operations is tricky because of rounding. Bit-wise operations are even more difficult to reason about. Fortunately, it is exceedingly rare for app code to branch on the results of these operations. Instead, we have observed that trusted library code is far more likely to branch on the results of these operations. As long as we can ensure that all library outputs are explicit, then we do not need to solve CSPs involving difficult operations when in trusted code.

**Arrays.** Dex provides instructions for inserting (`iput`) and retrieving (`iget`) data from an array. Due to type-conversion problems, SpanDex does not allow tainted indexing of non-numeric arrays. In particular, an app may not use a tainted variable to index into an Object array.

Handling an `iget` operation requires keeping a checkpoint of the array in the op-dag node for the variable holding the result. For example, say that all of the entries in an int array are zero or one, and that an app indexes into the array with a tainted variable. The returned value would be stored in a tainted variable. If the app later branched on the tainted variable, then SpanDex must look at the array checkpoint to determine which indexes would have returned the same value as the executed `iget`. In practice, tainted `iget` instructions are rare, and when they do occur the arrays are small.

Unlike a tainted `iget`, a tainted `iput` instruction is dangerous. Consider an attacker that initializes an array $a$ with known size, such that all entries are equal to zero. It then stores the first password character in the variable $s$ and inserts a one into $a[s]$. Because SpanDex maintains per-entry labels for arrays, $a[s]$ is tainted, but no other entries are. The attacker can then incrementally send each value in the array to its server: only $a[s]$ is tainted and will be stopped by SpanDex. Unfortunately, stopping the app at this point is too late, since the number of received zeros reveals the value of $s$. As a result of this attack, tainted `iput` instructions are illegal.

Finally, Dex also provides instructions such as `filled-new-array` for creating and populating arrays, and SpanDex disallows tainted operands on these instructions.

### 2.4.4 Trusted Libraries

As described above, there are a number of operations on tainted data that would add significant complexity to SpanDex's CSP solver to support. Even worse, the complexity of the op-dags that combinations of these operations would create make it doubtful that even a sophisticated solver could handle them quickly, if at all. Ideally, these operations would never arise, and if they did, an app would never branch on their results. Sadly, this not the case. Many apps require cryptographic and string-encoding libraries to handle passwords, and these libraries are rife with difficult to handle operations as well as branching on the results of those operations.

Trying to solve such complex CSPs would make SpanDex unusable: non-malicious apps would halt just trying to encrypt a password. At the same time, ignoring flows generated by these operations is not secure. Luckily, we have observed that branching on the results of difficult operations consistently occurs within a handful of simple platform libraries.

Thus, SpanDex's approach to handling difficult implicit flows is to identify the functionality that creates them in advance and to isolate these flows inside trusted implementations. As long as the outgoing information flows from these libraries are always tainted and explicit, SpanDex does not need to worry about their internal control-flow leaking secret information. Furthermore, this code is open and well known, is protected by the Java type system, and can be modified to eliminate implicit flows through the library API.

The set of libraries that SpanDex trusts not to leak information implicitly is shown in Listing 2.1.

```
java.lang.String (selected methods excluded)
java.lang.Character
java.lang.Math
java.lang.IntegralToString
java.lang.RealToString
java.lang.AbstractStringBuilder
java.net.URLEncoder
java.util.HashMap
android.os.Bundle
android.os.Parcel
org.bouncycastle.crypto
```

Listing 2.1: Set of libraries trusted by SpanDex to not leak information implicitly.

Nearly all of this code is either stateless string encoding and decoding, or cryptography.

### 2.4.5 Various Attacks and Counter-measures

We described several attacks in Section 2.3.2 that are beyond the scope of SpanDex. In this section, we describe several other attacks and how SpanDex might handle them.

First, SpanDex does not allow tainted data to be written to the file system or copied to another process via IPC. This is reasonable because mobile apps should only require a user's password to retrieve an OAuth token from a remote server. After receiving the token, the app should discard the user's password. If an app tries to copy tainted data to an external server, then SpanDex must consult the domains in the set of reachable op-dag root nodes.

Second, an attacker could have multiple apps under its control generate multiple overlapping (but not identical) p-sets. Each individual p-set would appear safe, but when combined at the attacker's server, they could collectively reveal an unsafe amount of information. Relatedly, a malicious app could request a user's password multiple times and compute different ranges on each password copy.

One way to detect this class of attacks is by inspecting the membership of a secret's p-sets. For the apps that we have observed, p-sets usually correspond to natural character groupings, e.g., numbers, lower-case letters, upper-case letters, and related special characters. P-sets containing unusual character groupings could be a strong signal that an app is malicious.

The solution to this attack suggests a larger class of counter-measures that use information from the p-sets and op-dag to detect malicious behavior. For example, anomalous operation mixes or an unusually large op-dag could indicate an attack. One of the advantages of SpanDex is that it gives the monitor a great deal of insight into how an app operates on password data. We believe that this information could enable a rich universe of policies, though enumerating all of them is beyond the scope of this paper.

Finally, it is possible that SpanDex is vulnerable to certain classes of side-channel and timing attacks that we have not considered. However, any attack that relies on branching on tainted data would be detected. For example, consider the well-known attack on Tenex's password checker [23]. Even though the attack uses a page-fault side channel that is out of SpanDex's scope, SpanDex would have prevented it because each additional character comparison would have narrowed its p-set to an unsafe level.

## 2.5 Implementation

Our SpanDex prototype is built on top of TaintDroid for Android 2.3.4. We modified TaintDroid to support p-sets and op-dags, and made several modification to the Android support libraries. Most of our changes to these libraries were made in java.lang.String.

First, public String methods whose return value could reveal something about a tainted string's value are not considered trusted to ensure that p-sets are updated properly (e.g., equals(Object), compareTo(String)).

Second, as a performance optimization, the Dalvik VM replaces calls to certain performance-critical Java methods with inlined "intrinsic" functions that are written in C and built in to the Dalvik VM (e.g., String.equals(Object), String.compareTo(String)). However, if an intrinsic inlined function operates on a tainted string and performs comparisons involving the string's characters, we are unable to update the p-sets accordingly. To avoid this, we modified Dalvik's intrinsic inlines that operate on strings to check if the string is tainted and, if so, invoke the Java version instead.

Third, Android's implementation of java.lang.String performs an optimization when converting an ASCII character to its String value: it uses the character's ASCII code to index into a constant char array containing all ASCII characters. If the character to be converted is tainted, we prevent this optimization from being used, as it would result in an array lookup with a tainted index.

Finally, we modified the android.widget.TextView and implemented a custom IME with a special tainted input mode that can be enabled to indicate to SpanDex when a sequence of characters is sensitive (i.e., a password).

## 2.6 Evaluation

In order to evaluate SpanDex, we sought answers to the following questions: How well does SpanDex protect users' passwords from an attacker? What is the performance overhead of SpanDex?

### 2.6.1 Password Protection

As described in Section 2.3.2, we have designed SpanDex based on an attacker that has access to a large list of cleartext passwords. The attacker knows that a user's password is in the list, and uses untainted information from its malicious app to narrow a user’s password to a smaller set of possibilities. To understand how well SpanDex can protect users from such an attack, we need to know the kind of p-sets that real apps induce, we need access to a large list of cleartext passwords, and we need a realistic distribution of how passwords are used. All of these pieces of information will allow us to calculate the number of expected logins an attacker would need to guess a user’s password, given the amount of untainted password information that SpanDex allows apps to reveal.

First, we ran 50 popular apps from Google’s Play Store. Each of these apps required a login, and we used the same 35-character password for each app. The password contained one lower-case letter (‘a’), one upper-case letter (‘A’), one number (‘0’), and one of each of the 32 non-space special ASCII characters. 42 ran without modification[1]. The top row of Table 2.1 shows each character in the password.

Eight apps invoked native code before requesting a user’s password[2]. While these apps would have to be modified to run under SpanDex, waiting to invoke native code before requesting a user’s password is unlikely to require major changes. All other apps ran normally.

Table 2.1: Password-character p-set sizes for 42 popular Android apps

| | ! | ” | # | $ | % | & | ’ | ( | ) | * | + | , | - | . | / | 0 | : | ; | < | = | > | ? | @ | A | [ | \ | ] | ˆ | _ | ‘ | a | { | \| | } | ˜ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Max | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 | 95 |
| 75th | 90 | 16 | 33 | 90 | 90 | 33 | 33 | 90 | 90 | 90 | 90 | 90 | 90 | 90 | 83 | 92 | 90 | 90 | 33 | 90 | 92 | 90 | 90 | 90 | 90 | 32 | 65 | 90 | 90 | 90 | 95 | 90 | 90 | 90 | 90 |
| Med | 16 | 12 | 12 | 16 | 16 | 12 | 12 | 16 | 16 | 13 | 16 | 16 | 13 | 13 | 14 | 10 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 26 | 6 | 6 | 6 | 6 | 6 | 6 | 90 | 4 | 4 | 4 | 4 |
| 25th | 12 | 4 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 1 | 12 | 12 | 1 | 1 | 12 | 10 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 26 | 5 | 5 | 5 | 5 | 1 | 5 | 26 | 4 | 4 | 4 | 4 |
| Min | 1 | 1 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 3 | 4 | 1 | 1 | 1 | 10 | 1 | 1 | 1 | 1 | 1 | 1 | 3 | 26 | 1 | 1 | 2 | 4 | 1 | 4 | 26 | 1 | 1 | 3 | 4 |

For the 42 apps that ran unmodified, after their password was sent, we inspected the p-set for each password character and counted its size. Table 2.1 shows the

[1] Audible, Amazon, Amazonmp3, Askfm, Atbat, Badoo, Chase, Crackle, Ebay, Etsy, Evernote, Facebook, Flipboard, Flixster, Foursquare, Heek, Howaboutwe, Iheartradio, imdb, LinkedIn, Myfitnesspal, Nflmobile, Pandora, Path, Pinger, Pinterest, Rhapsody, Skout, Snapchat, Soundcloud, Square, Tagged, Textplus, Tumblr, Tunein, Twitter, Walmart, Wordpress, Yelp, Zillow, Zite, and Zoosk

[2] Dropbox, Hulu+, Kindle, Mint, Skype, Spotify, Starbucks, and Voxer

maximum, 75th percentile, median, 25th percentile, and minimum p-set size for each password character. The header of the table shows the password. The first thing to notice is that the p-sets for the letters in our password (i.e., ‘A’ and ‘a’) were never smaller than 26. This makes sense, since each app is branching to determine that the character is either a lower or upper case letter. The same is true for the number in our password, ‘0’. No numeric p-set was smaller than 10.

The more difficult cases are the non-alphanumeric special characters. For these cases, the p-sets are fairly app specific. In some cases, the app’s control flow depends on a specific character (e.g., Skout with several special characters), but most characters’ p-sets remain large across most apps. With the exception of ‘*’, ‘-’, ‘.’, and ‘_’, all non-alphanumeric characters had large p-sets for 75% of apps or more.

Given this observed app behavior, we next obtained the uniqpass-v11 list of 131-million unique passwords [24]. The list contains passwords from a number of sources, including the Sony Pictures [22] and Gawker leaks [21]. To simulate an attack, we selected a password, $p$, from the list and computed the p-sets that a typical app would generate for $p$. In particular, we assume that the attacker can infer $p$’s length and whether each character is a lower-case letter (26 possibilities), an upper-case letter (26 possibilities), a number (10 possibilities), or a member of a block of special ASCII characters (i.e., the 16 characters below ‘0’, the 7 characters between ‘9’ and ‘A’, the 6 characters between ‘Z’ and ‘a’, and the 4 characters after ‘z’).

This information gave us a kind of regular expression for $p$ based on the type of each of its characters. We call the set of passwords matching this expression the *match set* and the size of the match set the *match count*. The larger a password’s match count, the more uncertain an attacker is about what password the user entered. We computed the match count for all passwords in the uniqpass list in this way. Finally, we counted the number of passwords with a given match count to arrive at the inverse distribution function.

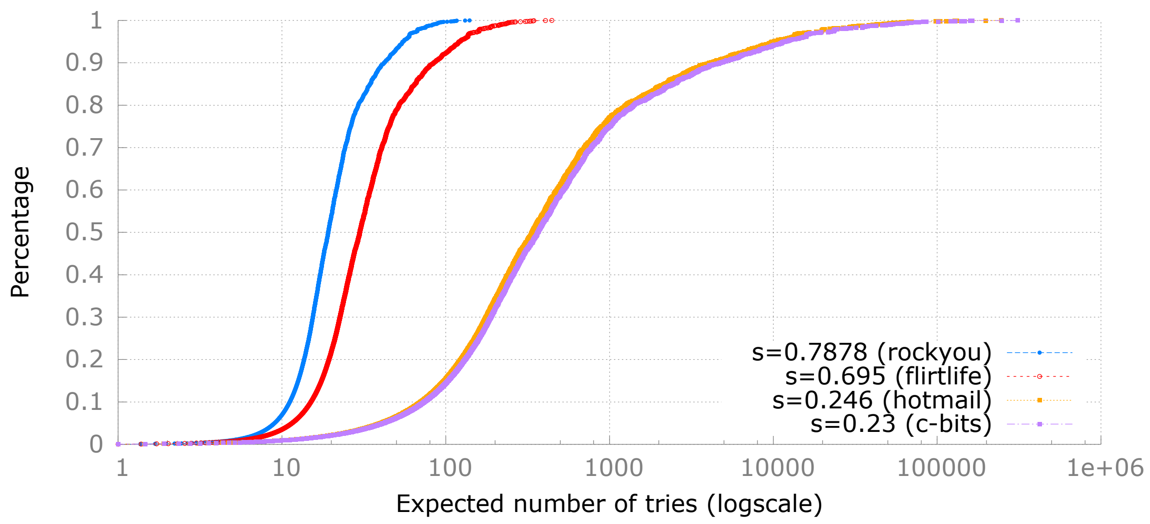

FIGURE 2.3: CDFs of expected login attempts using the uniqpass password list.

These calculations show that if SpanDex allows an attacker to learn the p-sets for a password from a typical app, the attacker will have trouble narrowing the set of possible passwords for the user. In particular, 92% of passwords have a match count greater than 10,000, 96% of passwords have a match count greater than 1,000, 98% of passwords have a match count greater than 100, and 99% of passwords have a match count greater than 10.

Unfortunately, recent work on a variety of password databases suggest that password usage follows a zipf distribution [25]. Thus, we also model the $N$ passwords in a match set as a population of $N$ elements that contains exactly one success (as a user would only have one correct password). Next, we let $n$ be the random variable denoting the number of tries required to guess the correct password and find $E[n]$, the expected value of $n$. If the passwords are all equally probable, we try them in random order. Otherwise, we try them in the descending order of their probability. Note that each password try is done without replacement, i.e., after trying $i$ passwords, we only consider the remaining $(N-i)$ passwords when picking the next most probable password.

A study of the distribution of passwords publicly leaked from Hotmail, flirtlife.de, computerbits.ie, and RockYou found that the passwords in each of these sets can be reasonably modelled by a zipf distribution with $s$ parameter values of 0.246, 0.695, 0.23, and 0.7878 respectively [25]. Using these values of $s$, we modeled the passwords in each match set and computed the CDF of $E[n]$.

When $s = 0.7878$, 95% of the time, the attacker is likely to guess the correct password within 50 tries. When the $s$ value for the zipf distribution is 0.246 or less, 99% of passwords are expected to require 10 or more login attempts, and 90% of passwords are expected to require 80 or more attempts. for all users. Figure 2.3 shows the CDFs for all four $s$ values.

Unfortunately, we do not know the usage distribution for the uniqpass dataset since it contains only unique passwords.

### 2.6.2 Performance Overhead

To measure the performance overhead of SpanDex we used the CaffeineMark benchmark and compared it to stock Android 2.3.4 and TaintDroid. Both TaintDroid and SpanDex ran without any tainted data. Since SpanDex only handles password data that is discarded after an initial login, this is SpanDex's common case. The benchmark was run on a Nexus S smartphone. The results are in Figure 2.4.

Overall, SpanDex performs only 16% worse than stock Android and 7% worse than TaintDroid. Stock Android performs significantly better than either TaintDroid or SpanDex in the string portion of the benchmark. This is because TaintDroid and SpanDex both disable some optimized string-processing code to store labels.

Finally, we would like to note that when testing apps in Section 2.6.1, we did not encounter any noticeable slow down under SpanDex. This was due to login being dominated by network latency and the simplicity of the CSPs these apps generated.

## 2.7 Conclusion

SpanDex tracks implicit flows by quantifying the amount of information transferred through implicit flows when an app executes a tainted control-flow operation. Using a strong attacker model in which a user's password is known to exist in a large password list, we found that for a realistic password-usage distribution, for 90% of users an attacker is expected to need 80 or more login attempts to guess their password.

## 2.8 Acknowledgments

This chapter draws on the peer-reviewed work published at USENIX Security 2014 [26] and I would like to thank the anonymous reviewers for their helpful comments.

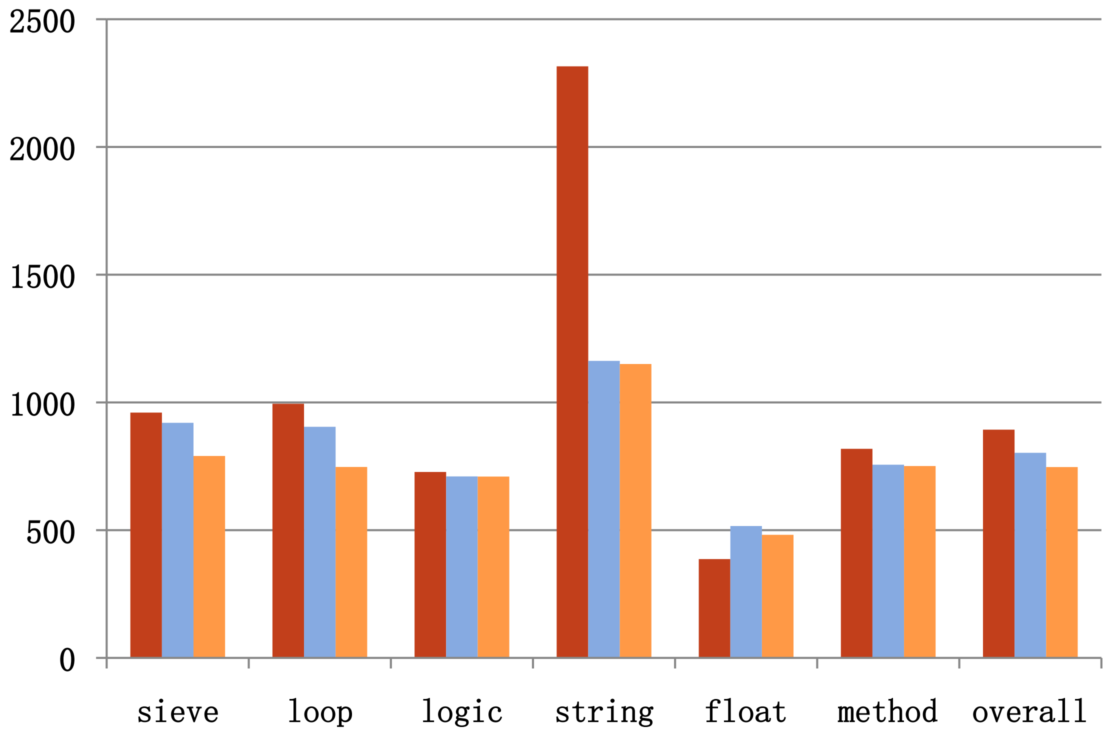

FIGURE 2.4: CaffeineMark results for Android (left bar), TaintDroid (middle bar), and SpanDex (right bar).

SpanDex has seen the involvement of a group of people over the years of which I am fortunate to have been a part of. I would like to thank my co-authors, Landon P. Cox, Peter Gilbert, Ali Razeen, Bi Wu, Geoffrey Lawler and Sai Cheemalapati. My main contribution has been in evaluating SpanDex on real-world apps. The SpanDex work was supported by Intel through the ISTC for Secure Computing at UC-Berkeley as well as the National Science Foundation under NSF awards CNS-0747283 and CNS-0916649.

# Chapter 3

# TaintTrap: Native Code Taint Tracking

## 3.1 Introduction

Android apps are typically written in Java and compiled to Dalvik Bytecode interpreted by the Dalvik Virtual Machine (VM). However, apps can also leverage native code that executes outside of the VM. For performance or legacy reasons, apps can provide their own shared libraries within the application package, written in C/C++, which execute directly as native code. A recent study found that 16% of apps use native code [27]. Furthermore, starting with Android 4.4, a new experimental Android runtime (ART) [28] aims to replace the default Dalvik runtime, and provides automatic and transparent ahead-of-time compilation of Dalvik code to native code for apps. Furthermore with the release of Android 5.0, ART is the default runtime, leading towards 100% native code execution in apps.

Hence, we believe supporting native code is key to providing complete system-wide information-flow tracking. Unfortunately there is no existing smartphone tracking system that supports native code tracking without significant performance impact on the device.

TaintDroid [6], the current state of the art Android on-device taint tracking system, is only capable of monitoring taint within the Dalvik VM. If the Dalvik code attempts to execute an app provided native function, TaintDroid is unable to continue to propagate taint until the native function returns. For this reason, TaintDroid's policy is to block all 3rd party apps from loading and invoking packaged native libraries. Unfortunately, this strong restriction can break any application that relies on native code, a common occurrence for the popular Android apps which rely on native code for performance reasons.

Another recent system, NDroid [27], also notes the importance of native code and extends TaintDroid to support native code interactions. However, it does not run on a real smartphone limiting its applicability, and relies on QEMU for virtualization, preventing apps from interacting with real environments (e.g., GPS or camera).

Motivated by ART and limitations of native code taint tracking for mobile devices, our goal is to create a practical taint tracking system that runs on a real smartphone device and allows apps to freely use native code, while maintaining the same tracking capabilities of Dalvik code offered by TaintDroid.

The intuition behind this work is based on the observation that apps, while trusted with sensitive data (e.g. passwords, contacts, GPS, camera), access this data or perform computations on it either rarely or for brief periods of time during the entire usage of the app. Using this insight we aim for a smartphone specific hardware-software approach that is built around the common case: apps rarely touch tainted data.

The key idea is to only perform the expensive taint tracking in rare cases when needed, otherwise providing the ability to run with negligible performance overhead. The purpose of our system is to deliver seamless end-to-end taint tracking across the Dalvik VM interpreter, platform native libraries and native libraries used by applications.

The proposed system has the following objectives:

- Full-system taint tracking on Android, spanning both Dalvik VM and native ARM code.
- Ability to switch between Unmodified Native Execution and Emulation, only when strictly needed.
- Low-overhead fine-grained Taint Access Faults by leveraging hardware support.

Section 3.2 provides an overview of the system. Section 3.3 covers the implementation details of our prototype. We evaluate the system in Section 3.4, review related work in Section 3.5, and conclude in Section 3.6.

## 3.2 System Overview

Since continuously instrumenting all native code execution on Android is prohibitively slow, we envision a flexible hybrid approach between a Dalvik taint tracking system and a native code tracking system. We extend TaintDroid's existing VM taint tracking with native code taint tracking and allow native code to execute unmodified so long as it does not access any tainted data. The challenge is knowing when native code touches tainted data while letting the code run unmodified. The key idea is that a mechanism for protecting access to tainted memory is sufficient to allow code to execute unmodified, relying on the protection to interrupt (trap) execution if the program does access tainted data.

We propose TaintTrap, a taint tracking system where native code tracking is designed with the ability to switch between unmodified/non-tainted execution and tainted execution. A key benefit of TaintTrap is Selective Emulation: the ability to emulate on-demand only the instructions accessing tainted data. This is a significant benefit in the common case of apps infrequently accessing tainted data (e.g. one-time password login) as the majority of app's execution is unaffected by TaintTrap.

Figure 3.1 provides a high level overview of the native code execution phases in TaintTrap's design.

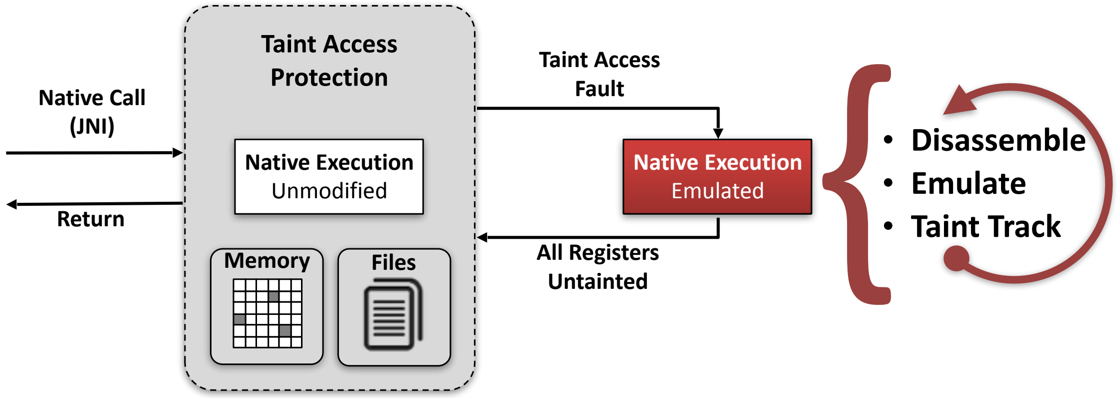

FIGURE 3.1: TaintTrap overview of on-demand native code taint tracking using emulation.

In this section, we describe the essential components needed for full-system taint tracking used by TaintTrap.

### 3.2.1 Dalvik VM Tainting

The VM interpreter executes Dalvik bytecode and for each bytecode opcode, additional instrumentation is added for taint propagation logic responsible for updating shadow memory and for assigning per-file taint tags. This instrumentation is part of the existing TaintDroid system.

### 3.2.2 Taint Map

At any given time, TaintTrap is aware of the range of addresses containing tainted data. This is achieved by keeping track of tainted data movement together with instrumenting system library functions such as `libc`'s `read` and `write`. The memory taint map keeps track of tainted data stored in memory. For each word of memory, it keeps track of its associated 32-bit taint tag. Whenever an instruction that references memory is emulated, the taint map is accessed to either check if the memory being read is tainted or to update the taint tag when memory is written. Separate from the memory taint map, TaintTrap also keeps track of taint for each architectural register.

### 3.2.3 Taint Access Protection

TaintDroid's tracking loses visibility when apps call into native code. The role of the TaintTrap's Taint Access Protection (TAP) layer is to ensure that once control is transferred from the VM to native code, any memory access to data inside the Taint Map triggers a *Taint Access Protection Fault.* Upon a TAP Fault, control is then transferred to a *TAP Handler* responsible for preparing the transition to *Native Execution Emulation.* To also intercept any access to tainted files on the device and trigger a TAP Fault, we also need to make transparent changes to the file system interface.

### 3.2.4 Native Execution Emulation

In emulation, we execute the application binary with complete (registers, memory and I/O) taint propagation logic. The taint tracking aware binary can either be statically instrumented with taint tracking instructions, dynamically in the form of JIT or taint tracking is emulated. Static instrumentation can be useful if the requirement is to instrument the entire binary, while TaintTrap's approach instruments only when strictly needed, on-demand depending on the data the program accesses. Thus, statically instrumenting the binary is not a feasible option, particularly since it requires statically determining which code to instrument and code touching tainted data could vary at run-time.

To emulate any instruction, TaintTrap requires dynamic disassembly and decoding of assembly instructions. For this task, TaintTrap leverages `darm` [29], an efficient structural disassembler for ARM.

For performance reasons, it is important that TaintTrap spends as little time as needed performing emulation. The key is determining what conditions are needed to allow a transition back to unmodified native execution. The insight comes from observing with a mechanism such as TAP that is able to protect tainted memory accesses, once architectural registers are no longer tainted, it is safe to switch to unmodified native execution.

### 3.2.5 Trust and Attacker Model

TaintTrap's emulation framework is implemented in user-space and relies on non-privileged OS features that the application being instrumented has access to. The operating system (Android platform and Linux kernel) is considered trusted. This means that TaintTrap can be subverted by a malicious adversary with specific knowledge of the system's internals. For example, an attacker with knowledge of emulation state memory layout within the app's virtual memory, could craft a malicious app that can change emulation state to disable monitoring or covertly un-taint tainted data by directly writing memory inside the app's own address space. However, TaintTrap is still useful for monitoring malicious apps that are not intentionally subverting TaintTrap, although in this case we cannot make any guarantees.

Upon decoding an instruction, `darm` provides detailed structural information. A listing of the most commonly used fields is shown in Listing 3.1.

```
struct darm_t {
    // the original encoded instruction
    uint32_t        w;
    // the instruction label
    darm_instr_t    instr;
    darm_enctype_t  instr_type;
    // conditional execution flags, if any
    darm_cond_t     cond;
    // flag: conditional flags updated
    uint32_t        S;
    // flag: add or to subtract the immediate
    uint32_t        U;
    // flag: pre-indexed or post-indexed addressing
    uint32_t        P;
    // flag: write-back
    uint32_t        W;
    // flag: immediate set
    uint32_t        I;
    // rotation value
    uint32_t        rotate;
    // register operands
    // dest, 1st operand, 2nd, accum, transferred, 2nd transferred
    darm_reg_t      Rd, Rn, Rm, Ra, Rt, Rt2;
    // immediate operand
    uint32_t        imm;
    // register shift info
    darm_shift_type_t shift_type;
    darm_reg_t      Rs;
    uint32_t        shift;
    // some instructions operate on bits and specify bits used
    uint32_t        lsb, msb, width;
    // bitmask of registers affected by STM/LDM/PUSH/POP instr
    uint16_t        reglist;
} darm_t;
```

Listing 3.1: Snippet of `darm`'s decoded instruction (most common elements).

## 3.3 Implementation

TaintTrap is a prototype emulation framework built on TaintDroid 4.1.1, and is implemented with 4000 lines of C code, not including the `darm` disassembler. One of our primary objectives is to create a working prototype capable of running directly on a smartphone, using existing software primitives and hardware. Building a prototype provides a great level of understanding and insight into the fundamental challenges of a practical system. This section outlines the implementation decisions, tradeoffs and opportunities for improvement.

### 3.3.1 Emulation Required Hooks

For emulation to function correctly, it requires knowledge of when a process starts or exits, and when new threads are created or freed. This is achieved by adding hooks to a select set of library calls, e.g. thread creation and teardown. Android uses a process called Zygote that has been initialized and has all the core libraries linked in. When a new application is started, a new VM is created by forking the Zygote process, an operation that saves time and memory as libraries are shared. Given that core libraries are read-only, at any given time there is a single copy in memory. The select set of library calls is shown in Listing 3.2.

```
pid_t forkAndSpecializeCommon(const u4* args, bool isSystemServer)
void __thread_entry(int(*func)(void*), void *arg, void **tls)
void __bionic_clone_entry(int (*fn)(void *), void *arg)
void __bionic_atfork_run_child()
void _pthread_internal_free(pthread_internal_t* thread)
void _exit_thread(int retCode)
```

Listing 3.2: Hooked platform functions needed by emulation.

With the exception of Zygote's fork implementation in `dalvik.system.Zygote`, all other functions are part of `libc`'s `Pthreads` implementation and used by public APIs (e.g. `pthread_create` and `pthread_exit`). In our implementation, we modified the platform source code (considered trusted code) to insert needed hooks, however it may be possible to dynamically hook functions with a mechanism like `LD_PRELOAD` offered by the dynamic linker.

### 3.3.2 Memory Taint Map

The taint map needs to cover both the stack and the rest of a program's memory such as text and data segments, dynamically allocated memory via `mmap`, `sbrk` or `malloc`. For performance reasons, TaintTrap linearly allocates the taint map from anonymous virtual memory through `mmap` using the flags `MAP_ANONYMOUS` and `MAP_NORESERVE` which allows allocating more virtual memory than physical memory. Accessing the taint tag for a given memory address is fast and requires directly indexing into the taint map based on the offset computed from the given address and the starting memory address covered by the taint map. While the taint maps need to cover both current and potential future memory expansion, only a very small fraction of memory is expected to become tainted and thus needing additional physical memory compared to the program's original memory requirements. If the available virtual memory is insufficient for allocating the taint maps, alternative approaches can trade off speed for smaller memory footprint: bitmaps and on-demand page level taint storage allocation. In practice, TaintTrap allocates separate taint map for stack memory. A lookup step at runtime determines the matching taint map based on the desired address and information stored with each taint map identifying its type and covered address range. A sample memory layout with allocated taint tag storage is shown in Listing 3.3.

```
10000000 - 18000000 // Taint Tags (Stack)
20000000 - 40000000 // Text + Data
40000000 - 70000000 // Heap
70000000 - be800000 // Taint Tags (Non-Stack)
be800000 - bf000000 // Stack
ffff0000 - ffff1000 // Vectors
```

Listing 3.3: Example of program memory layout including taint tag storage.

The structure of a taint map is shown below in Listing 3.4:

```
struct taintmap_t {
    uint32_t    *data;              /* mmap-ed data */
    uint32_t     start;             /* address range start */
    uint32_t     end;               /* address range end */
    uint32_t     bytes;             /* (end - start) bytes */
    taintpage_t *pages;             /* taintpages */
};
```

Listing 3.4: Emulator taint map data structure.

### 3.3.3 Architectural Registers' Taint

The register taint map is implemented as a 16-element array for fast lookups and indexing is performed with the register number: `r0` to `r15`. Taint propagation rules are similar to an instructions' data movement. For example the instruction `add r0, r1, r2` performs `r0` $\leftarrow$ `r1 + r2` and `Taint(r0) = Taint(r1)` $\cup$ `(r2)`, where $\cup$ represents the union of two taint tags, which in our implementation is a bitwise OR, same as in TaintDroid.

### 3.3.4 Protection Granularity

The mechanism we use to enforce tainted data access protection is traditional OS Page Protections. Due to the coarse grain page granularity (4KB), we expect false positives. Prior work on hardware support for fine grained memory protection in the form of watchpoints [30] is a very promising match for TaintTrap as a way of removing such false positives. The impact of protection granularity is measured in detail in the evaluation section.

### 3.3.5 Protection Primitive

To change page protections TaintTrap uses the OS provided `mprotect` system call, which allows changing the memory access protection for a region of memory. For example, `mprotect(0x1000, PAGE_SIZE, PROT_NONE)` prevents any memory access to the page located at address `0x1000`. Any subsequent access to a tainted page triggers a segmentation-fault signal (`SIGSEGV`). Conversely, instead of `PROT_NONE`, to restore original protections after a page is no longer tainted, a combination of one or more of `PROT_READ`, `PROT_WRITE` and `PROT_EXEC` allows a mix of read, write memory access and code execution of the page.

### 3.3.6 Tainting Memory

Whenever TaintTrap taints a piece of memory, if the data resides on a untainted page, the page is immediately protected from subsequent memory accesses. This ensures that if any other thread of the application tries to access the newly tainted data, a TAP Fault triggers, providing the entry-point for emulation.

### 3.3.7 Untainting Memory

A memory read or write reference can only access and taint or untaint 1, 2 or 4 bytes at a time. Instructions such as `PUSH/POP`, `LDM/STM` that load and store multiple values are considered as making multiple memory references, one for each element. However, protecting memory at the page-level prevents directly knowing if all the non-written bytes on a page are non-tainted. It is prohibitively expensive to naively scan all other 1023 words in the taint map. Instead, to efficiently detect when a page is tainted and when it first becomes untainted, TaintTrap keeps a per-page taint counter with values ranging from 0 to 1024, representing the number of word-size tainted elements on a 4KB page. When the counter is first incremented from 0 to 1, the page is tainted and `mprotect` is called. Similarly, when the counter is decremented from 1 to 0, the page becomes untainted, and `mprotect` is called to restore the original page protections.

### 3.3.8 Special Cases

User-space emulation and taint tracking exposes a number of cases requiring special handling. In this section we discuss some of the most interesting cases we observed.

**Tainted Stack.** On ARM the Stack Pointer (SP) can be used as a general purpose register. Thus, it is possible for the SP to become tainted. In fact, we have seen a legitimate switch statement on tainted data in jpeg exif benchmark. The switch statement switches on tainted data read from a file. Assembly instructions modify the SP using a logical shift with the tainted value as immediate. This is a legitimate use although it is more closely related to implicit flows. TaintTrap only emulates explicit flows and hence ignores taint propagation using the SP.

Another stack special case is the stack itself becoming tainted and hence protected from access. This can happen by legitimate register spills on the stack or during function calls where registers are saved and restored. Saving a tainted register on the stack is sufficient to protect the entire stack page around the value. This is not a correctness issue but a performance one, since it results in additional work due to false taint access traps when the remaining stack page is used, requiring TaintTrap to check the faulting instruction for taint, single-step it and resume execution back unmodified native code. A high number of watchpoints as proposed by prior work [30], can alleviate the performance hit of false taint access traps.

**Tainted Thread-Local Storage.** The Linux Pthreads implementation stores the Thread-Local Storage (TLS) data at the top of the stack. TLS is used for internal state such as Thread ID and `errno`, or it can accessed through Pthreads APIs such as `pthread_key_create` and `pthread_{set,get}specific`. Android's Bionic `libc` library allocates storage for up to 64 TLS slots of 32-bit each and the rest of the page is used for stack frames. Hence, if the first stack page becomes tainted, accessing TLS causes a TAP Fault. This is problematic since emulation also needs to access TLS, which causes an infinite trap loop. To avoid trapping on the TLS data, we allocate 1024 TLS slots, thus ensuring any tainted stack page does not overlap TLS data. As this issue stems from a protection granularity limitation, watchpoints are again a useful primitive to have that TaintTrap can benefit from.

**Trampolines.** Since TaintTrap only emulates user-level instructions and since it is not always running due to the nature of selective emulation, it needs some assistance from the platform. TaintTrap relies trampolines to check for tainted data its behalf and notify back if necessary. Consider an app that reads and writes a file on the filesystem. The operations performed are `read` and `write` syscalls on a file descriptor. If the file is tainted and TaintTrap is not already running for the thread performing the syscall, it cannot notice the taint interaction. To solve this problem, TaintTrap inserts trampolines in `libc`'s `read` and `write` functions (which already exist as wrappers around the syscalls). The trampolines then transfer control to code which checks if the source or destination buffers are tainted and if the file descriptor is associated with a tainted file.

While trampolines add the required taint checks, they also expose a subtle issue. Consider the potential callers for `read/write`:

1. App while executing without emulation (common case). This can be easily handled by making sure the taint logic checks for a thread-local running flag that is set while emulation is running. When the flag is not set (no emulation underway), trampolines are responsible for checking taint, otherwise trampoline taint checks are skipped as they are performed during instruction emulation.
2. App while under emulation. Since the app is already under emulation when it calls `read/write`, the taint propagation logic part of the trampoline gets emulated. While not a correctness issue, this is a major performance issue that TaintTrap addresses. The problem is complicated by the stack potentially being tainted and trapping on access, preventing any trampoline taint check bypass logic from executing as any subsequent function call would trap. TaintTrap's workaround leverages the fact that for emulation, each thread has a dedicated alternate stack that is never protected. Thus, instead of running

the bypass logic on the regular stack, it is run on the TaintTrap stack through the use of a TLS-based side channel. Trampolines check if the current stack frame is tainted/protected before attempting to run any taint checks, and if tainted, request it be performed on the dedicated stack by recording what taint checking function needs to be called then raising a trap using a specially crafted undefined instruction that is delivered as a `SIGILL` to TaintTrap. This ensures that TaintTrap can distinguish between a common `SIGSEGV` on tainted data access triggering emulation, versus a request to call a function on its stack and resume execution.

3. TaintTrap for its own use (not requested by app). This can only happen due to TaintTrap's distinct usage of libraries, for example writing the execution instruction trace. In this case TaintTrap never directly manipulates an app's tainted data and hence trampoline taint checks need to be skipped. One solution is to make sure TaintTrap does not call these trampolines directly by avoiding use of any libraries used by the app (e.g. `libc`) and instead relying on syscalls directly.

**Accessing Tainted Memory.** While page protections prevent an app from accessing tainted data by forcing emulation, they unfortunately prevent emulation itself from accessing memory on tainted pages. If TaintTrap directly performs loads and stores within a tainted page, it causes another `SIGSEGV` (TAP Fault), leading to an infinite loop. While emulation can use `mprotect` to give back rights, perform the load or store, then restore restricted access, it requires two syscalls for each memory operation making it expensive. However, a correctness issue arises since other non-emulated threads could temporarily access a tainted page while one emulated thread is manipulating the protections and data for the same page. For this reason, TaintTrap always keeps tainted pages protected from access but uses an additional mechanism to be able to read and write memory without issuing user-level loads and stores. The Linux kernel provides a `procfs` mechanism under `/proc/self/mem` that allows an app to read its own memory using standard `read` and `write` syscalls on a file descriptor. This is used by debuggers like GDB to set and change software breakpoints.

The code in Listing 3.5 shows how to `read` a value at address `ptr` and return it in `val`. Note that opening the file descriptor is a one-time operation. We can aggregate the `lseek` and `read` syscalls into a single `pread` syscall, which allows reading a file descriptor at a given offset. Similarly we use `pwrite` instead of `lseek` and `write`. A code snippet that uses `pread` is shown in Listing 3.6.

```
int fd = open("/proc/self/mem", O_RDWR); // one-time cost
lseek(fd, ptr, SEEK_SET);
read(fd, &val, sizeof(val));
```

Listing 3.5: Using `/proc/self/mem` to read memory: two syscalls per access.

```
int fd = open("/proc/self/mem", O_RDWR); // one-time cost
pread(fd, &val, sizeof(val), offset);
```

Listing 3.6: Using `/proc/self/mem` to read memory: one syscall per access.

From a security perspective, the `/proc/self/mem` mechanism does not give an app more privileges than it already has, it allows an app to read and write its own memory, which it is able to regardless of the mechanism. If a page is protected, a malicious adversary can unprotect the page before accessing it with loads and stores.

The kernel implementation of `pread/pwrite`, requires finding a free page, calling `copy_from_user` if handling a `pwrite`, `access_remote_vm`, and `copy_to_user` if handling `pread`. In this case `access_remote_vm` is a very expensive operation. Normally the function is used to access any process' virtual memory, although in our case we are accessing memory in the calling process.

The kernel's implementation is shown in Listing 3.7. The kernel uses copy to/from functions since it is not allowed to directly read user-level memory. The special functions leverage the Memory Management Unit (MMU) and account for faults due to invalid memory mappings or permissions.

The current limitation on accessing tainted and protected memory exposes an opportunity: TaintTrap may avoid the kernel workaround by protecting memory at the user-level but emulating the hypervisor level with direct access to tainted memory, avoiding costly tainted memory accesses.

```
ssize_t mem_rw(struct file *file, char __user *buf,
        size_t count, loff_t *ppos, int write)
{
    struct mm_struct *mm = file->private_data;
    unsigned long addr = *ppos;
    ssize_t copied;
    char *page;
    if (!mm)
        return 0;
    page = (char *)__get_free_page(GFP_TEMPORARY);
    if (!page)
        return -ENOMEM;
    copied = 0;
    if (!atomic_inc_not_zero(&mm->mm_users))
        goto free;
    while (count > 0) {
        int this_len = min_t(int, count, PAGE_SIZE);
        if (write && copy_from_user(page, buf, this_len)) {
            copied = -EFAULT;
            break;
        }
        this_len = access_remote_vm(mm, addr, page, this_len, write);
        if (!this_len) {
            if (!copied)
                copied = -EIO;
            break;
        }
        if (!write && copy_to_user(buf, page, this_len)) {
            copied = -EFAULT;
            break;
        }
        buf += this_len;
        addr += this_len;
        copied += this_len;
        count -= this_len;
    }
    *ppos = addr;
    mmput(mm);
free:
    free_page((unsigned long) page);
    return copied;
}
```

Listing 3.7: Kernel code of `pread/pwrite` on Android Linux Kernel 3.0.31.

**Tainted Atomic Sequences.** ARM does not provide a single atomic instruction that can update memory. Instead it offers a general synchronization primitive that splits atomically updating memory into two steps with the help of exclusive monitors. First, the `LDREX` (Load-Exclusive) instruction loads a word from memory and updates the exclusive monitor to track the synchronization operation. Second, the STREX (Store-Exclusive) instruction conditionally stores a word to memory. If the executing processor's exclusive monitor permits, the store updates memory and returns the value 0, otherwise it does not update memory and returns the value 1.

Consider the pseudocode shown in Listing 3.8 that performs an atomic increment. While the atomic increment code performs as expected in a non-emulated environment, it is possible that under emulation the page containing the ptr address is tainted and hence protected from memory access. If the page is tainted, upon attempting to execute the LDREX instruction, a TAP Fault triggers emulation but unfortunately the only way to emulate a synchronization primitive is with another primitive. If TaintTrap's emulation attempts to execute a crafted LDREX and STREX to the same address, it faults again causing an infinite loop. To avoid the delicate situation, TaintTrap instead uses a per-thread Pthreads mutex that is only locked and unlocked during emulation, while memory is read and written using /proc/self/mem's pread and pwrite instead (which does not fault even if the page is protected). The mutex is locked before emulating the LDREX and unlocked after emulating the STREX.

```
int32_t
__bionic_atomic_inc(volatile int32_t* ptr) {
    int32_t prev, tmp, status;
    do {// assembly
        ldrex prev, [ptr]
        add   tmp, prev, 1
        strex status, tmp, [ptr]
    } while (status != 0);
    return prev;
}
```

Listing 3.8: Atomic increment from libc using LDREX/STREX primitive.

Emulation of tainted atomic sequences suffers from a similar limitation to the one previously seen when accessing tainted memory. The same approach of performing memory accesses and emulation at the hypervisor level may avoid the special case handling currently required and reduce overheads.

**Prefetching.** Any emulated prefetched instructions, `PLD` (preload data) and `PLI` (preload instruction), are likely to no longer have their desired benefit due to emulation interference. Hence, TaintTrap ignores prefetch instructions and treats them as `NOP`s, without having any impact on execution correctness.

**Page Protection Permissions.** To restore page permissions when a page becomes untainted, TaintTrap needs to save the page permissions at the time when the page was last tainted. This is necessary since protecting a tainted page requires removing all existing permissions. Unfortunately, Linux does not provide an efficient mechanism for a user-level application to query the permissions for specific page. Instead the kernel offers a `procfs` based mechanism which requires reading and parsing a text file representation located at `/proc/self/maps`. This file is updated by the kernel when memory is allocated for a process. Using this file method is very slow and incorrect for multi-threaded applications since the file cannot be locked for changes while it is being read.

A solution is to either 1) extend the kernel to provide page protection query, or 2) keep track of memory allocations performed via `mmap` and `sbrk` system calls and any non-emulation specific `mprotect` calls that could change these protections. TaintTrap favors the latter approach instead of changing the kernel.

### 3.3.9 Prototype Limitations

TaintTrap's prototype implementation has a few non-fundamental limitations as it does not currently have emulation support for:

- Vector SIMD instructions. We observed these instructions in some Bionic `libc` optimizations such as `memcpy` and disabled SIMD support by undefining `__ARM_HAVE_NEON`.

- Vector Floating Point (VFP) in 3rd party apps. System libraries that did use VFP were recompiled with `gcc`'s `-mfloat-abi=soft` to enable software emulation. Bionic `libc` required undefining `__ARM_HAVE_VFP`.

- Coprocessor instructions. In system library code we have observed a single use of these instructions: accessing the Thread ID inside TLS. We only support this special case.

## 3.4 Evaluation

The main questions we want to answer are: 1) does TaintTrap work and propagate taint for native code?; and 2) can TaintTrap be used on a real device? The current TaintTrap prototype, even with its limitations, runs on a Galaxy Nexus smartphone and is able to detect, intercept and propagate taint for a set of programs. In this section we explore the main causes of overhead in the current implementation of TaintTrap, discuss opportunities for improvement and use a performance model to gauge their benefit if applied to TaintTrap.

### 3.4.1 Benchmarks

We consider three micro-benchmarks to gauge TaintTrap's overhead: `password`, `matrix`, and `jpeg-exif`:

- `password`, represents a minimal example of leaking a user's password through a native code file system write call. It is used to test tracking functionality across files. TaintTrap captures and flags the leak by tainting the file with the stolen password. Listing 3.9 shows the micro-benchmark code.

- `matrix`, simulates variable sized tainted data and computation through the use of a classic three loop matrix multiply ($A \times B = C$) and with adjustable taint, ranging from tainting one row to all rows of the matrix $A$. TaintTrap propagates taint to the corresponding rows of the result matrix $C$. Measurements are gathered for the inner loop and do not include the allocation, initialization and initial one-time tainting of the input.

- `jpeg-exif`, is based on an observed native code use case seen in the Instagram app, where after the user takes a camera picture, it is saved to disk and processed using native code. Instagram first uses `libjhead` library to parse and extract the extended JPEG metadata (EXIF), and then applies a custom image filter. We reproduced the parsing of the JPEG EXIF phase as standalone micro-benchmark. TaintTrap intercepts the use of a tainted image file and tracks tainted data usage throughout the parsing phase. Unlike the first two benchmarks, `jpeg-exif` includes `libc` `read/write` as well as atomic sequences, which are handled correctly by TaintTrap.

```
// Application Dalvik VM Code (Java)
public boolean authenticate() {
    // platform trusted API for input
    String password = user_input();
    // untrusted 3rd party app code
    return native_authenticate(password);
}
```

```
// Application Native Code (C/C++)
bool native_authenticate(char *password) {
    hide(password);
    return login(password);
}
void hide(char *password) {
    char stash[MAX_LEN];
    int c = 0;
    while(password[c] != '\0') {
        // obfuscate/hash password
        stash[c] = password[c] - 32;
        c++;
    }
    // stash password on filesystem
    int fd = open("leak.tmp", O_CREAT|O_RDWR);
    write(fd, stash, c);
    close(fd);
}
```

```
# Example password
$ echo 't0psecr3t!@#' | xxd
0000000: 7430 7073 6563 7233 7421 4023 0a         t0psecr3t!@#.
# Obfuscated password (subtract 0x20)
$ xxd cache.tmp
0000000: 5410 5053 4543 5213 5401 2003            T.PSECR.T. .
```

Listing 3.9: Password leak test case.

A comprehensive set of statistics for each of the benchmarks is shown in Table 3.1. For each of the 3 benchmarks, a comparison is made between Selective Emulation (SE), where TaintTrap performs taint tracking and emulation only as needed, and Full Emulation (FE), where TaintTrap is continuously running from start to finish. The main observation is that coarse grain memory protection results in a high number of expensive taint traps, up to 96% for `jpeg-exif`. In this case most of the false traps are caused by protecting the stack due to one or more tainted values being temporarily left on the stack. Since the stack is frequently used during execution, false traps result in a major cause of performance overhead. For `matrix`, a comparison of compiler optimizations levels (O0 vs O3) shows how in the optimized case (O3), the matrix inner loop is tuned such that at any given time there is at least a tainted register, preventing TaintTrap from turning itself off in the SE case. In the unoptimized O0 case, SE can turn itself off but half of the traps are false taint traps due to protection granularity.

Table 3.1: TaintTrap comparison of Full Emulation (FE) and Selective Emulation (SE) with per event breakdown.

| Event | **Benchmark Count** | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | **password** | | **matrix (O0)** | | **matrix (O3)** | | **jpeg-exif** | |
| | SE | FE | SE | FE | SE | FE | SE | FE |
| emulated instructions | 124 | 149 | 859,297 | 1,180,265 | 227,815 | 227,883 | 172,855 | 241,291 |
| emulated load instructions | 17 | 20 | 561,248 | 672,048 | 112,794 | 112,827 | 24,732 | 30,554 |
| emulated store instructions | 13 | 13 | 68,671 | 69,699 | 5,243 | 5,262 | 13,179 | 14,270 |
| emulated atomic sequences | - | - | - | - | - | - | 22 | 30 |
| traps | 6 | - | 136,351 | - | 3 | - | 13,379 | - |
| traps on stack access | 5 | - | 69,792 | - | 0 | - | 12,654 | - |
| traps non-stack hit % | 16.7 | - | 24.0 | - | 33.3 | - | 0.4 | - |
| traps stack hit % | 0 | - | 0.8 | - | 0 | - | 2.7 | - |
| traps non-stack miss % | 0 | - | 24.8 | - | 66.7 | - | 5.1 | - |
| traps stack miss % | 83.3 | - | 50.4 | - | 0 | - | 91.9 | - |
| memory read refs total | 29 | 25 | 561,248 | 672,048 | 112,802 | 112,827 | 44,814 | 51,723 |
| memory write refs total | 28 | 21 | 68,671 | 69,699 | 5,243 | 5,262 | 35,469 | 37,620 |
| memory read refs tainted | 12 | 12 | 65,536 | 65,536 | 32,768 | 32,768 | 14,508 | 14,502 |
| memory write refs tainted | 9 | 9 | 33,792 | 33,792 | 1,024 | 1,024 | 10,053 | 10,047 |
| trampoline read total | 0 | 0 | - | - | - | - | 3 | 3 |
| trampoline write total | 1 | 1 | - | - | - | - | 55 | 55 |
| trampoline stack taint | 0 | 0 | - | - | - | - | 7 | - |
| trampoline taint read | 0 | 0 | - | - | - | - | 3 | 3 |
| trampoline taint write | 0 | 0 | - | - | - | - | 53 | 53 |
| page protection updates | 1 | 1 | 2,049 | 2,049 | 10 | 10 | 54 | 54 |
| tainted reg instructions | 24 | 24 | 131072 | 131072 | 65536 | 65536 | 35059 | 35059 |
| tainted mem instructions | 12 | 12 | 33792 | 33792 | 1024 | 1024 | 10965 | 10959 |
| tainted instructions % | 38.7 | 32.2 | 19.2 | 14.0 | 29.2 | 29.2 | 26.7 | 19.1 |

## 3.4.2 False Positives

Instead of emulating the entire program (Full Emulation) to add taint tracking, TaintTrap's Selective Emulation design aims to only emulate instructions that are required for taint tracking correctness and revert to native/unmodified execution when no longer necessary. There are two scenarios that can artificially increase TaintTrap's number of instructions that require emulation:

1. Taint trap false positives. Due to page protection granularity, when only part of a page is tainted and a memory access is performed on the non-tainted region, a trap can still be raised, requiring TaintTrap to check if tainted data is accessed and to single-step (emulate) the faulting instruction.
2. Tainted registers. To ensure tracking correctness, although the instruction being emulated may not access and propagate taint, TaintTrap is unable to revert to unmodified native code while any CPU registers are marked as tainted.

To measure the impact of these two cases, we used the `matrix` micro-benchmark with dynamically allocated matrix storage. First, for each matrix row memory is allocated through `malloc`, resulting in sequential memory allocations in our case and thus being susceptible to taint trap false positives. Second, `mmap` is used instead for allocation, resulting in dedicated pages for each row, thus removing any taint trap false positives when an entire row is tainted. As a lower bound, the ideal smallest number of instructions emulated is determined by counting only the instructions that strictly propagate taint through registers and memory. However, achieving this in a real system is impractical.

The results are shown in Figure 3.2, comparing the emulated instruction counts for Selective Emulation. Sequential allocation results in false positive traps while allocating with `mmap` ensures each row resides on a dedicated page, removing false positive traps. Ideal (lower bound) strictly represents only the instructions that propagate taint.

The less amount of taint, the higher the impact of coarse grain protection false traps, while at the far end when the entire input is tainted, TaintTrap's Selective Emulation ability is unused and the system becomes equivalent to performing Full Emulation.

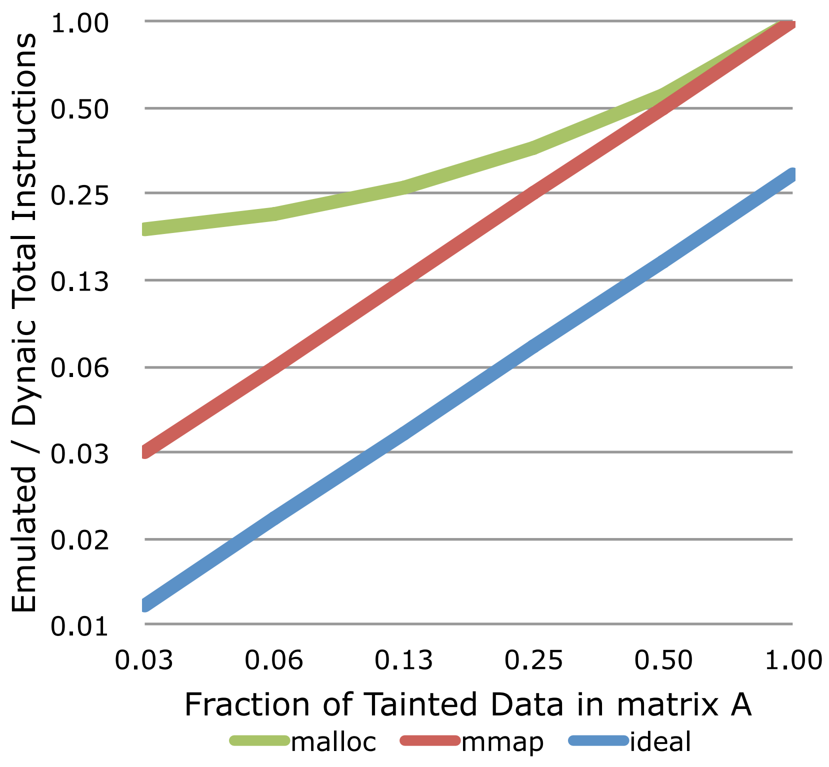

FIGURE 3.2: Impact of false positives due to memory protection granularity depending on sequential allocation (`malloc`) versus page-aligned (`mmap`).

### 3.4.3 TaintTrap Overheads

To better understand the underlying components of TaintTrap we measured the overheads of system operations such as signals, traps and `pread/pwrite`, relative to a null syscall. The overheads are shown in Figure 3.3 where each overhead is the average of at least 10000 operations. As seen, our workaround for accessing tainted data through the help of the kernel's `pread/pwrite` incurs significant overhead. In our measurements, we find a noticeable difference between the `pread` and `pwrite` times, where writing is more expensive than reading. This is possibly caused by the read path being cached while the write requires additional work to flush and propagate the new data.

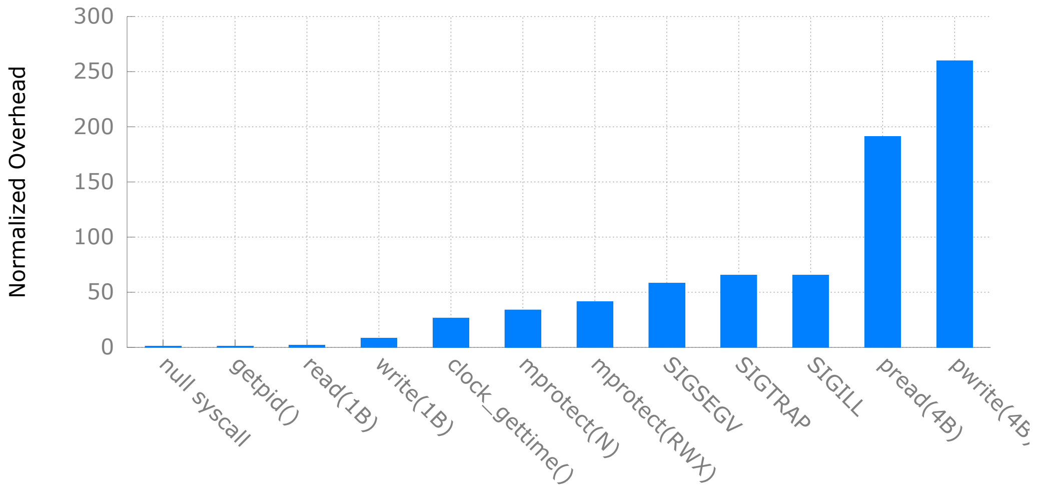

Figure 3.3: System average overhead for various operations compared to null-syscall.

To better see the impact of the using `pread/pwrite` for accessing tainted data, we measure the breakdown between user and kernel time for our `jpeg-exif` benchmark. The results are shown in Figure 3.4. TaintTrap's workaround for accessing tainted memory through kernel help (`/proc/self/mem`), significantly impacts performance seen in the increased kernel time fraction.

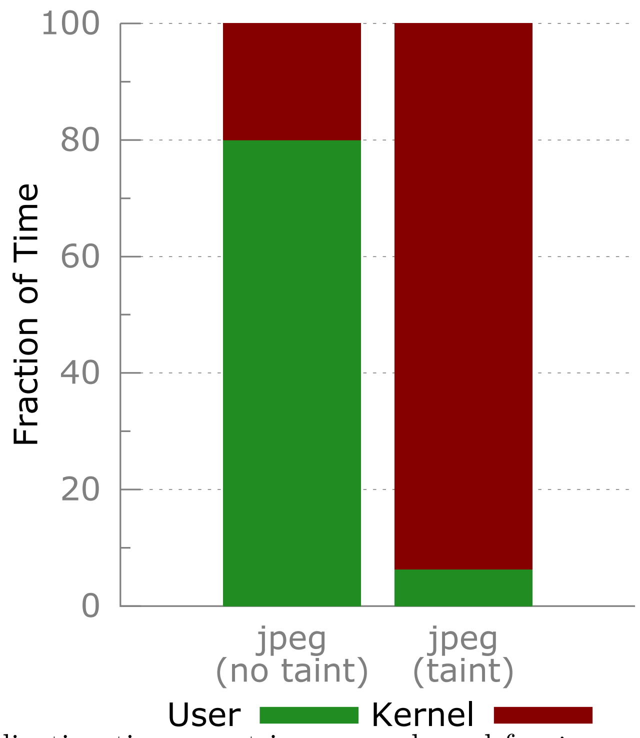

FIGURE 3.4: Application time spent in user or kernel for `jpeg-exif` without taint tracking (left) and with taint tracking (right).

Another important metric in understanding TaintTrap's behavior is found by observing the total amount of tainted memory throughout a program's execution. As seen in Figure 3.5, about 15-40% of the entire memory references are for tainted data. This is to be expected given the nature of our micro-benchmarks focusing on accessing and manipulating tainted data. However, in a real app use case, the fraction of tainted memory references is expected to be much lower as is likely dominated by regular non-tainted flows.

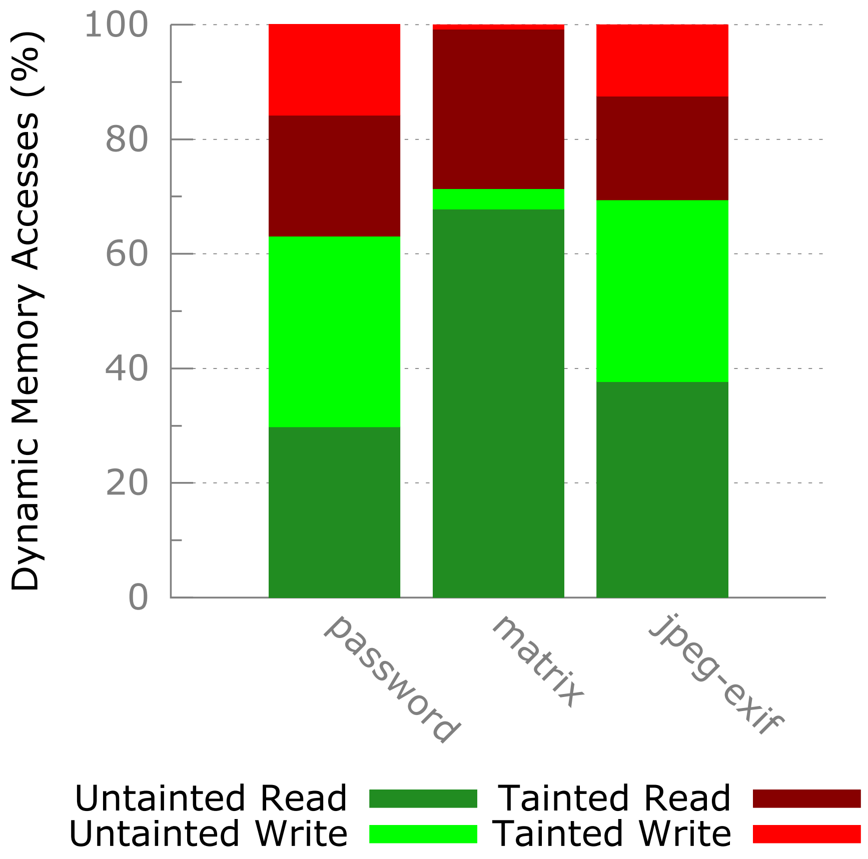

FIGURE 3.5: Memory references for tainted data over entire execution.

### 3.4.4 Accelerating Taint Tracking

One of TaintTrap's implementation objectives is to run on existing devices, and to achieve this with existing software support. However, it is important to consider how TaintTrap may evolve in future systems. For example, eliminating taint access false-positives and the resulting overhead, TaintTrap can take advantage of Dune [31], which exposes privileged CPU features, including page tables, to user-level applications. This can significantly reduce the performance overhead of managing and protecting tainted data. Although TaintTrap makes use of existing memory protection (`mprotect`), the coarse protection granularity can be detrimental to performance. However, prior work on fine-grained memory protection through watchpoints [30], can be very beneficial to TaintTrap's approach. Watchpoints can be used efficiently used and modified directly from user-code. Additional hardware support in the form of range caches provide great performance for watchpoint operations [32]. Coupled with user-level traps, TaintTrap's selective emulation can be very practical for many real world use cases involving sensitive data on mobile devices, without having to sacrifice performance and battery life for the common case of apps not interacting with sensitive data. For additional security, TaintTrap could run at the hypervisor level. A hypervisor helps isolate the emulator and its state from the program being instrumented, and provides strong guarantees for tracking malicious apps without the risk of TaintTrap being subverted.

### 3.4.5 Performance Model

To understand the impact of potential improvements to TaintTrap, we consider a simple performance model where overheads can be adjusted separately. For example, user-level faults can significantly reduce the trap overhead, a hypervisor accessing taint no longer requires expensive `pread/pwrite` operations and hence lowers the overhead of accessing tainted and protected memory.

The starting point for the model comes from benchmarking individual TaintTrap operations, shown in Table 3.2. For example, $T_{Emu}$ represents the average overhead of emulating an instruction, including disassembly time $T_{Disassembly}$ which represents half the time. Changing the memory protection bits for a page (4K) using `mprotect` is represented by $T_{Protect}$. Taking a taint trap results in handling a `SIGSEGV` which is represented by $T_{Trap}$.

More formally, we model the execution time as follows:

Modeled Execution Time (MET) =

(# traps) $\times$ ($T_{Trap}$) +

(# mem read + # mem write) $\times$ $T_{CheckTaintPage}$ +

Table 3.2: Measured average cycle overheads for TaintTrap internal operations.

| **Event** | **Symbol** | **Cycles** |
|---|---|---|
| Processor cycles per instruction | $T_{CPI}$ | 1.25 |
| Read tag for single address | $T_{GetTaintTag}$ | 25 |
| Check if page is tainted | $T_{CheckTaintPage}$ | 20 |
| Disassemble one instruction | $T_{Disassembly}$ | 100 |
| Emulate one instruction without taint | $T_{Emu}$ | 200 |
| Emulate one instruction with memory taint | $T_{EmuTaint}$ | 20,000 |
| Read tainted memory using `pread` | $T_{GetTaintMem}$ | 40,000 |
| Write tainted memory using `pwrite` | $T_{SetTaintMem}$ | 70,000 |
| Allocate temporary buffer (4K) | $T_{AllocBuffer}$ | 8,000 |
| Change memory protection bits (4K) | $T_{Protect}$ | 10,000 |
| Taint Access Protection (TAP) Fault | $T_{Trap}$ | 12,000 |
| Check if memory range is tainted (4K) | $T_{GetTaintBuffer}$ | 4,000 |
| Write taint for memory range (4K) | $T_{SetTaintBuffer}$ | 7,600 |
| Check if file descriptor is tainted | $T_{GetTaintFile}$ | 200,000 |
| Taint file descriptor | $T_{SetTaintFile}$ | 210,000 |

(# taint mem read + # taint mem write) $\times$ $(T_{TaintAccess} + T_{TaintMemAccess})$ +

(# page protection updates) $\times$ $T_{Protect}$ +

(# instructions) $\times$ $(T_{CPI} + T_{Emu})$

The MET above includes the execution time without emulation: (# instructions) $\times$ $(T_{CPI})$.

We explore four scenarios, each subsequent one making additional improvements:

1. Selective Emulation + Taint: The current implementation of TaintTrap using the expensive `pread/pwrite` for tainted memory access. The exact overheads are taken from the measurements seen in Table 3.2 with the exception that we model read and writes as taking the same time, specifically $T_{TaintMemAccess}$ is $T_{SetTaintMem}$.

2. Fast Taint: Quick access to tainted data is crucial to system overhead. From prior work, we model the availability of watchpoints [30] and consider $T_{TaintMemAccess}$ and $T_{Trap}$ as 20 cycles.

3. Fast Emulation: Another major component to TaintTrap overheads is the emu-

lation overhead per instruction, $T_{Emu}$. The current prototype does not yet take advantage of typical improvements such as caching the disassembly, emulating basic blocks instead of single instructions or saving and re-using the emulation work with Just-In-Time (JIT) compilation. We model such improvements as $T_{Emu}$ per instruction taking 20 cycles, although we believe this can be more aggressively reduced.

4. Selective Emulation (Ideal): As a lower-bound, we model emulation of bare minimum instructions (only instructions that propagate taint), although this is not practical to implement particularly for tight loops like our `matrix` micro-benchmark which can add additional overheads to enable/disable emulation for each iteration, which we do not model.

The performance model is applied using profiling information from the `matrix` micro-benchmark, using O3 optimizations. The information includes how many instructions need emulation and access tainted data. The performance model system overhead estimates are shown in Figure 3.6. Each of the four scenarios are compared across variable taint ratios of the input matrix $A$. For the `matrix` benchmark, tainting 100% of matrix $A$ results in an overall dynamic taint access fraction of almost 30%.

The most interesting data point is the Fast Emulation variant, representing a version of TaintTrap that improves on the existing selective emulation by leveraging watchpoints together with more efficient emulation. We find the results to be very encouraging, as we expect the common case for apps to have dynamic fractions of tainted data accessed around or well under 1%. For example, the password leak is under 200 instructions, a negligible amount compared to over 1 billion instructions a second executed by the processor (Galaxy Nexus runs at 1.2GHz and most ARM instructions have an average CPI of 1 cycle). We believe TaintTrap can be practical

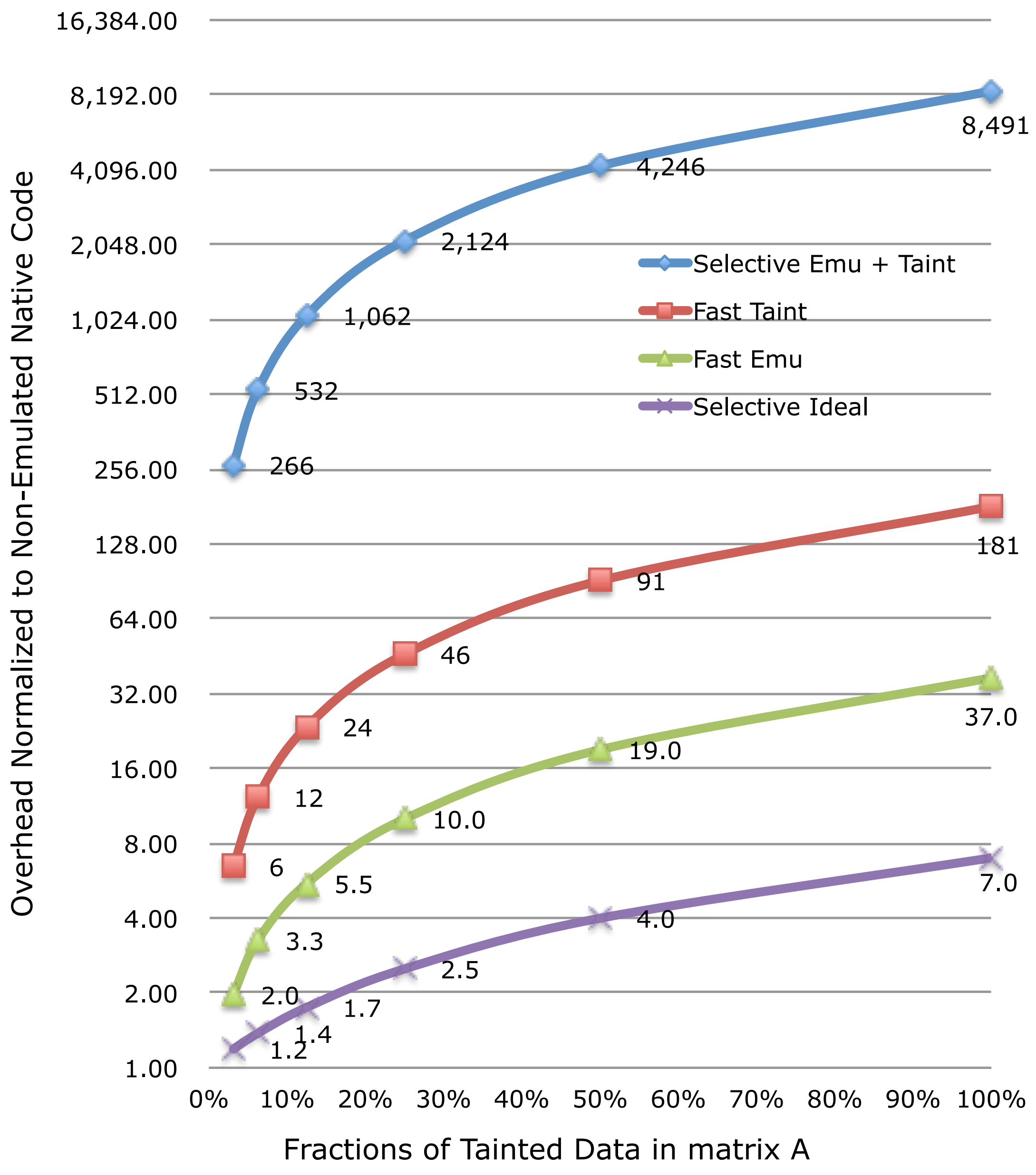

FIGURE 3.6: Performance model: emulation overhead improvements across different fractions of `matrix` $A$ taint ratios.

and very useful for running on real mobile devices, where it efficiently runs in the rare case when needed, and does not impact the perceived user experience, in the common case when apps do not interact with the user's sensitive data.

## 3.5 Related Work

Dynamic taint analysis (DTA) has been an active research area since the pioneering work on information-flow tracking by Denning [1] in the 70s. There are three main approaches used to implement taint tracking: instrumentation-based, interpreter-based and architecture-based. We discuss each of these below and focus on the flexibility and overhead of taint tracking.

**Instrumentation-based systems.** These systems generally leverage one of the available dynamic binary-rewriting frameworks available: Pin [33–35], Valgrind [36], Dynamo, DynamoRIO, StarDBT. A limitation of using these frameworks is that they only support process level taint tracking.. The approach is limited to user-level applications and instrumentation overhead is generally significant, prohibiting real time usage on a smartphone. Systems that provide full-system taint tracking use virtualization or emulation such as offered by Xen or QEMU, also have significant overheads [27, 37–40].

Most systems only provide support for 1-bit tags but still incur a significant overhead [7, 41]. Other systems make use of very architectural specific features to improve overheads but end up suffering from lack of wide-spread applicability or restrictions. Minemu [42] implements a fast taint-tracking x86 process-based emulator which supports one byte taint tags for each byte of emulated process memory. Although it achieves low overheads between 1.5x and 3x by using SSE registers to avoid register spills and store tags, it is specific to 32-bit x86 code and requires SSE extensions making their implementation non-portable and unable to support applications using

SSE instructions. LIFT [43] has 1-bit tags and extending to multi-bit tags significantly increases the memory overhead. LIFT uses the StartDBT binary translator which translates IA32 instructions to EM64T instructions since like with Minemu, EM64T has more registers than IA32, avoiding register spills for performance gains. Finally LIFT does not support multi-threading or implicit flows and does not work on 32-bit systems.

Like TaintDroid and TaintTrap, most systems only support explicit flows. Implicit flow support has been proposed in Dytan [8] for x86, which uses Pin making the overheads impractical for a mobile device. It leverages statically-computed post-dominance information and requires adding spurious definitions to make the memory definitions on both sides of a branch equal. The reported time overhead for data flows is 30x and including control flows it is 50x, while the space overhead is 240x. Another system, DTA++ [9] implemented on top of BitBlaze framework, targets implicit flows that can lead to *under-tainting* and uses symbolic execution but requires slow whole-system emulation. Other work has added support for limited types of implicit flows in the context of detecting security exploits [44].

Dynamic taint analysis has been used to determine privacy exposure [10, 45] as well as to track Internet worms [46]. Similar to TaintTrap's taint access protection, libdft [40] is built over Pin and uses page protections to block a tracked program from accidentally corrupting Pin or libdft. On-demand emulation and page protections have also been explored but in an environment with virtualization with Xen and emulation with QEMU [47].

**Interpreter- or VM-based.** TaintDroid [6] implements system-wide information-flow tracking with variable data object granularities ranging from one tag per variable or register to one tag per file. The tags are stored stored as a 32-bit bitvector and variable tags are adjacent in memory for spatial locality. It is specifically designed for the unique constraints of Android smartphones and thus taint propagation logic is implemented inside Android's Dalvik VM interpreter which makes it significantly more coarse grain than previous work where taint is done at the machine binary level. This has the benefit of very small overheads, 14-27% performance overhead and 3.5% memory overhead. TaintDroid however, is unable to monitor information-flows in native code.

Many language-specific optimizations have been proposed such as fine-grain character level taint tracking for Java [48]. Other systems have targeted JVM, PHP, Python and SQL. Laminar [49] offers decentralized information-flow control by providing language extensions for JVM and a security library. It also requires the help of programmer annotations which is not applicable in our environment.

**Architecture-based** Many hardware extensions and designs have been proposed to aid information-flow tracking. A majority only offer 1-bit tags for every physical memory word [50–53].

Raksha [52] is among the most configurable hardware DIFT implementations but does not support full-system information-flow (does not cover the operating system). Raksha supports tags per 4-bit tags per 32-bit where tags are 4-bit where each bit represents a security policy with distinct propagation rules and checks. Another key feature is that security exceptions are processed at the user-level avoiding an expensive switch to kernel mode. Whenever an exception is triggered, the hardware maintains its current privilege level and activates trusted mode . It then invokes the handler in the same address space as the application.

Tiwari et al. [32] address the performance critical component in all hardware DIFT architectures, the on-chip tag cache. Specifically they address the problem of storing and querying large multi-bit tags efficiently. They propose the Range Cache which provides support for fine-grain tracking up to the byte level, flexible tracking granularities with per-byte or per-word tags, and efficient on-chip storage of large multi-bit tags of 32-bit or larger. The work is based on the observation that tag bits exhibit a very high degree of spatial-value locality: large contiguous blocks of memory addresses store the same tag value. Thus, the Range Cache associates tags with ranges of addresses instead of individual addresses. Mondrian Memory Protection (MMP) [54] has been proposed to reduce tag storage overhead by offering a compact in-memory representation. It offers word-level granularity access control using a protection look-aside buffer. MMP is designed for tools that do not perform frequent updates. MMP similarly associates metadata with ranges of addresses but is limited to allocations defined directly through `malloc`, `free` or page boundaries. Another limitation of MMP is that ranges have to be aligned power-of-two sized which complicates the discovery of contiguous ranges as well as range split or merge operations. In contrast, Range Cache provides support for very fast range split, merge and update without any alignment restrictions.

One of the challenges that prevents wide-spread adoption of specialized hardware for dynamic software systems is that each of the different analysis types require highly specific hardware support. A generalized acceleration mechanism has a higher chance of being integrated in a commercial processor.

To address this opportunity, Greathouse et al. [30] argue for data watchpoints as a primitive useful in a large number of analyses including Dynamic Dataflow Analysis, Deterministic Concurrent Execution and Data Race Detection. In particular, they propose hardware support that provides software the ability to set virtually unlimited watchpoints. Watchpoints can be byte-accurate or cover any range of memory addresses. For performance reasons, they use two key mechanisms to enable on chip caching of watchpoints: a modified range cache (RC) proposed by Tiwari et al. [32] and a Watchpoint Lookaside Buffer (WLB). The RC handles large ranges of watchpoints efficiently but is not good for small watchpoint regions. To overcome that limitation a bitmap is used for compact representation and to avoid going out to memory to search the bitmap, the WLB is used to cache these accesses.

Dune [31] uses virtualization extensions to provide user-level access to privileged CPU features including the page table. This can be a very beneficial addition to TaintTrap. iWatcher [55] provides watchpoint-like support by leveraging thread-level speculation (TLS). TaintBochs [56] tracks sensitive information at the whole-system level with the goal of analyzing data lifetime. RIFLE [57] converts implicit to explicit flows with the help of a dynamic binary translation (DBT) by trading off coding restrictions. FlexiTaint [58] separates the storage of processing of taints from data. Minos [51] provides integrity bits and propagation logic that prevents control flow hijacking. Loki [59] aims to reduce the trusted computing base in a system and provides guarantees even in the presence of a compromised OS by using tagged memory support. Each 32-bit word is associated a 32-bit tag and a permissions cache is stores the set of tag values.

Parallel approaches have also been proposed: Speck [60] accelerates security checks using multiple cores. For taint analysis they support only 1-bit tags and in the best case report a 2x speedup using 8 cores relative to their own implementation of TaintCheck using Pin (which has 18x overhead). This places Speck clearly outside the space of real-time usage for smartphones. Ruwase et al. [61] propose a parallel yet relaxed DIFT implemented over a Log-Based Architecture (LBA). It only tracks flows through unary operations and supports 1-bit tags. Their work can reduce overhead to as low as 1.2x but requires 9 monitoring cores on 16-core chip multiprocessor and extensive hardware support for LBA, making it infeasible in a smartphone setting.

Other low-level hardware approaches have considered building a new design from the ground up and proposed gate-Level information-flow tracking [62] having the benefit of supporting explicit, implicit and timing flows. The tradeoff is in area increases of 70%, harder to program and is computationally less powerful than traditional processors.

Panorama [63] offers whole-system DIFT aiming to support off-line malware detection and analysis. The reported overhead is an average 20x and the system is too heavy-weight for a resource constrained smartphone. HDL have also been designed from the ground up with the goal of secure information-flow [64].

## 3.6 Conclusions

We extended the state of the art in dynamic information flow tracking on Android by creating a practical system built around on-demand native code emulation. TaintTrap is designed to leverage the common case for smartphone apps, where sensitive data is rarely and briefly accessed. To address performance overheads, we explored potential improvements building on prior work that could be very beneficial to our system. Given the results, TaintTrap can be practical for safeguarding user's sensitive data on a mobile device, without impacting the overall experience. Since TaintTrap works at the binary level, no source code or developer effort is required to support our system, allowing for simple deployment.

## 3.7 Acknowledgements

The TaintTrap work was supported by Intel through the ISTC for Secure Computing at UC-Berkeley as well as the National Science Foundation under NSF award CNS-0747283. I am very grateful to Jurriaan Bremer, for creating `darm`, the disassembler used by TaintTrap, and for being a great sounding board for ideas throughout TaintTrap's development. In our collaboration we have come up with a disassembler design that we feel is elegant, flexible and efficient.

# Bibliography

[1] Dorothy E. Denning. A Lattice Model of Secure Information Flow. *Commun. ACM*, 19(5):236–243, May 1976. ISSN 0001-0782. doi: 10.1145/360051.360056. URL `http://doi.acm.org/10.1145/360051.360056`.

[2] S. Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley Professional, 1999.

[3] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 281–294, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1301-8. doi: 10.1145/2307636.2307663. URL `http://doi.acm.org/10.1145/2307636.2307663`.

[4] Xuxian Jiang. Smishing Vulnerability in Multiple Android Platforms, 2012. URL `http://www.csc.ncsu.edu/faculty/jiang/smishing.html`.

[5] Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 95–109, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4681-0. doi: 10.1109/SP.2012.16. URL `http://dx.doi.org/10.1109/SP.2012.16`.

[6] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1924943.1924971`.

[7] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS '05, 2005.

[8] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 196–206, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-734-6. doi: 10.1145/1273463.1273490. URL `http://doi.acm.org/10.1145/1273463.1273490`.

[9] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2011.

[10] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS '11, San Diego, CA, February 2011.

[11] A. Sabelfeld and A. C. Myers. Language-based Information-flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, September 2006. ISSN 0733-8716. doi: 10.1109/JSAC.2002.806121. URL `http://dx.doi.org/10.1109/JSAC.2002.806121`.

[12] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and Event Processes in the Asbestos Operating System. In *Proceedings of the 20th ACM symposium on Operating Systems Principles*, SOSP '05, pages 17–30, New York, NY, USA, 2005. ACM. ISBN 1-59593-079-5. doi: 10.1145/1095810.1095813. URL `http://doi.acm.org/10.1145/1095810.1095813`.

[13] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information Flow Control for Standard OS Abstractions. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 321–334, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5. doi: 10.1145/1294261.1294293. URL `http://doi.acm.org/10.1145/1294261.1294293`.

[14] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making Information Flow Explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 19–19, Berkeley, CA, USA, 2006. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1267308.1267327`.

[15] Stephen McCamant and Michael D. Ernst. Quantitative Information Flow As Network Flow Capacity. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 193–205, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375606. URL `http://doi.acm.org/10.1145/1375581.1375606`.

[16] smali: An assembler/disassembler for Android's dex format, 2013. URL `http://code.google.com/p/smali/`.

[17] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. On the Limits of Information Flow Techniques for Malware Analysis and Containment. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pages 143–163, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-70541-3. doi: 10.1007/978-3-540-70542-0_8. URL `http://dx.doi.org/10.1007/978-3-540-70542-0_8`.

[18] Asia Slowinska and Herbert Bos. Pointless Tainting? Evaluating the Practicality of Pointer Tainting. In *Proceedings of the 4th ACM European Conference on Computer Systems,*, EuroSys '09, pages 61–74, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-482-9. doi: 10.1145/1519065.1519073. URL `http://doi.acm.org/10.1145/1519065.1519073`.

[19] Dongtao Liu, Eduardo Cuervo, Valentin Pistol, Ryan Scudellari, and Landon P. Cox. ScreenPass: Secure Password Entry on Touchscreen Devices. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 291–304, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1672-9. doi: 10.1145/2462456.2465425. URL `http://doi.acm.org/10.1145/2462456.2465425`.

[20] Apple. iPhone 5s: Touch ID, 2014. URL `http://images.apple.com/iphone/business/docs/iOS_Security_Feb14.pdf`.

[21] Laurie Segall. Gawker data exposed in major hack attack, 2010. URL `http://money.cnn.com/2010/12/13/technology/gawker_hacked/index.htm`.

[22] Peter Bright. Sony hacked yet again, plaintext passwords, e-mails, dob posted, 2011. URL `http://arstechnica.com/tech-policy/2011/06/sony-hacked-yet-again-plaintext-passwords-posted/`.

[23] Butler W. Lampson. Hints for Computer System Design. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, SOSP '83, pages 33–48, New York, NY, USA, 1983. ACM. ISBN 0-89791-115-6. doi: 10.1145/800217.806614. URL `http://doi.acm.org/10.1145/800217.806614`.

[24] Dazzlepod. Uniqpass v11, 2013. URL `https://dazzlepod.com/uniqpass/`.

[25] David Malone and Kevin Maher. Investigating the Distribution of Password Choices. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, pages 301–310, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1229-5. doi: 10.1145/2187836.2187878. URL `http://doi.acm.org/10.1145/2187836.2187878`.

[26] Landon P. Cox, Peter Gilbert, Geoffrey Lawler, Valentin Pistol, Ali Razeen, Bi Wu, and Sai Cheemalapati. SpanDex: Secure Password Tracking for Android. In *23rd USENIX Security Symposium*, Security '14, San Diego, CA, August 2014. USENIX Association. URL `https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/cox`.

[27] Chenxiong Qian, Xiapu Luo, Yuru Shao, and Alvin T. S. Chan. On Tracking Information Flows through JNI in Android Apps. In *Proceedings of the 44th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '14, Atlanta, Georgia, USA, June 2014.

[28] Google. Android Runtime (ART), 2014. URL `http://developer.android.com/preview/api-overview.html#ART`.

[29] Jurriaan Bremer. darm - Efficient ARMv7 Disassembler. URL `http://darm.re`.

[30] Joseph L. Greathouse, Hongyi Xin, Yixin Luo, and Todd Austin. A Case for Unlimited Watchpoints. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 159–172, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0759-8. doi: 10.1145/2150976.2150994. URL `http://doi.acm.org/10.1145/2150976.2150994`.

[31] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '12, pages 335–348, Hollywood, CA, 2012. USENIX. ISBN 978-1-931971-96-6. URL `https://www.usenix.org/conference/osdi12/technical-sessions/presentation/belay`.

[32] Mohit Tiwari, Banit Agrawal, Shashidhar Mysore, Jonathan Valamehr, and Timothy Sherwood. A Small Cache of Large Ranges - Hardware Methods for Efficiently Searching, Storing, and Updating Big Dataflow Tags. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 94–105, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-1-4244-2836-6. doi: 10.1109/MICRO.2008.4771782. URL `http://dx.doi.org/10.1109/MICRO.2008.4771782`.

[33] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. doi: 10.1145/1065010.1065034.

[34] Kim Hazelwood and Artur Klauser. A Dynamic Binary Instrumentation Engine for the ARM Architecture. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, pages 261–270, New York, NY, USA, 2006. ACM. ISBN 1-59593-543-6. doi: 10.1145/1176760.1176793. URL `http://doi.acm.org/10.1145/1176760.1176793`.

[35] Prashanth P. Bungale and Chi-Keung Luk. PinOS: A Programmable Framework for Whole-System Dynamic Instrumentation. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 137–147, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-630-1. doi: 10.1145/1254810.1254830. URL `http://doi.acm.org/10.1145/1254810.1254830`.

[36] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi: 10.1145/1250734.1250746. URL `http://doi.acm.org/10.1145/1250734.1250746`.

[37] Andrey Ermolinskiy, Sachin Katti, Scott Shenker, Lisa L Fowler, and Murphy McCauley. Towards Practical Taint Tracking. Technical Report UCB/EECS-2010-92, EECS Department, University of California, Berkeley, Jun 2010. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-92.html`.

[38] Lok Kwong Yan and Heng Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security '12, pages 29–29, Berkeley, CA, USA, 2012. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=2362793.2362822`.

[39] David (Yu) Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. TaintEraser: Protecting Sensitive Data Leaks Using Application-Level Taint Tracking. *SIGOPS Oper. Syst. Rev.*, 45(1):142–154, February 2011. ISSN 0163-5980. doi: 10.1145/1945023.1945039. URL `http://doi.acm.org/10.1145/1945023.1945039`.

[40] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, VEE '12, pages 121–132, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1176-2. doi: 10.1145/2151024.2151042. URL `http://doi.acm.org/10.1145/2151024.2151042`.

[41] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting. In *Proceedings of the 11th IEEE*

*Symposium on Computers and Communications*, ISCC '06, pages 749–754, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2588-1. doi: 10.1109/ISCC.2006.158. URL `http://dx.doi.org/10.1109/ISCC.2006.158`.

[42] Erik Bosman, Asia Slowinska, and Herbert Bos. Minemu: The world's fastest taint tracker. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, RAID'11, pages 1–20, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-23643-3. doi: 10.1007/978-3-642-23644-0_1. URL `http://dx.doi.org/10.1007/978-3-642-23644-0_1`.

[43] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 135–148, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2732-9. doi: 10.1109/MICRO.2006.29. URL `http://dx.doi.org/10.1109/MICRO.2006.29`.

[44] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proceedings of the 15th Conference on USENIX Security Symposium*, Security '06, Berkeley, CA, USA, 2006. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1267336.1267345`.

[45] Yu Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. Privacy Scope: A Precise Information Flow Tracking System For Finding Application Leaks. Technical Report UCB/EECS-2009-145, EECS Department, University of California, Berkeley, Oct 2009. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-145.html`.

[46] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the 20th ACM symposium on Operating Systems Principles*, SOSP '05, pages 133–147, New York, NY, USA, 2005. ACM. ISBN 1-59593-079-5. doi: 10.1145/1095810.1095824. URL `http://doi.acm.org/10.1145/1095810.1095824`.

[47] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. Practical Taint-based Protection Using Demand Emulation. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 29–41, New York, NY, USA, 2006. ACM. ISBN 1-59593-322-0. doi: 10.1145/1217935.1217939. URL `http://doi.acm.org/10.1145/1217935.1217939`.

[48] Erika Chin and David Wagner. Efficient Character-level Taint Tracking for Java. In *Proceedings of the 2009 ACM workshop on Secure Web Services*, SWS

'09, pages 3–12, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-789-9. doi: 10.1145/1655121.1655125. URL http://doi.acm.org/10.1145/1655121.1655125.

[49] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical Fine-Grained Decentralized Information Flow Control. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 63–74, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542484. URL http://doi.acm.org/10.1145/1542476.1542484.

[50] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-XI, pages 85–96, New York, NY, USA, 2004. ACM. ISBN 1-58113-804-0. doi: 10.1145/1024393.1024404. URL http://doi.acm.org/10.1145/1024393.1024404.

[51] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 221–232, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2126-6. doi: 10.1109/MICRO.2004.26. URL http://dx.doi.org/10.1109/MICRO.2004.26.

[52] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 482–493, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-706-3. doi: 10.1145/1250662.1250722. URL http://doi.acm.org/10.1145/1250662.1250722.

[53] Guru Venkataramani, Brandyn Roemer, Yan Solihin, and Milos Prvulovic. MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging. In *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 273–284, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 1-4244-0804-0. doi: 10.1109/HPCA.2007.346205. URL http://dx.doi.org/10.1109/HPCA.2007.346205.

[54] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian Memory Protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-X, pages 304–316, New York, NY, USA, 2002. ACM. ISBN 1-58113-574-2. doi: 10.1145/605397.605429. URL http://doi.acm.org/10.1145/605397.605429.

[55] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. iWatcher: Efficient Architectural Support for Software Debugging. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pages 224–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2143-6. URL `http://dl.acm.org/citation.cfm?id=998680.1006720`.

[56] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *Proceedings of the 13th USENIX Security Symposium*, Security '04, pages 22–22, Berkeley, CA, USA, 2004. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1251375.1251397`.

[57] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David I. August. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '37, pages 243–254, Washington, DC, USA, Dec 2004. IEEE Computer Society. ISBN 0-7695-2126-6. doi: http://dx.doi.org/10.1109/MICRO.2004.31.

[58] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. FlexiTaint: Programmable Architectural Support for Efficient Dynamic Taint Propagation. In *Proceedings of the 14th IEEE International Symposium on High Performance Computer Architecture*, HPCA '08, pages 173–184. IEEE, Feb 2008.

[59] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. Hardware Enforcement of Application Security Policies Using Tagged Memory. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 225–240, Berkeley, CA, USA, 2008. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1855741.1855757`.

[60] Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. Parallelizing Security Checks on Commodity Hardware. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 308–318, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-958-6. doi: 10.1145/1346281.1346321. URL `http://doi.acm.org/10.1145/1346281.1346321`.

[61] Olatunji Ruwase, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Shimin Chen, Michael Kozuch, and Michael Ryan. Parallelizing Dynamic Information Flow Tracking. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 35–45, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-973-9. doi: 10.1145/1378533.1378538. URL `http://doi.acm.org/10.1145/1378533.1378538`.

[62] Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. Complete Information Flow Tracking from the Gates Up. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '09, pages 109–120, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-406-5. doi: 10.1145/1508244.1508258. URL `http://doi.acm.org/10.1145/1508244.1508258`.

[63] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 116–127, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-703-2. doi: 10.1145/1315245.1315261. URL `http://doi.acm.org/10.1145/1315245.1315261`.

[64] Xun Li, Mohit Tiwari, Jason K. Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. Caisson: A Hardware Description Language for Secure Information Flow. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 109–120, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993512. URL `http://doi.acm.org/10.1145/1993498.1993512`.

[65] Sandeep R. Agrawal, Valentin Pistol, Jun Pang, John Tran, David Tarjan, and Alvin R. Lebeck. Rhythm: Harnessing Data Parallel Hardware for Server Workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14. ACM, March 2014. doi: 10.1145/2541940.2541956. URL `http://dx.doi.org/10.1145/2541940.2541956`.

# Biography

Valentin Pistol was born on April 7$^{th}$, 1986 in Craiova, Romania. He earned his B.S. in Computer Science from University of Craiova, Romania in 2009, M.S. in Computer Science from Duke University in 2011 and Ph.D. in Computer Science from Duke University in 2014.

His research interests include the design, implementation, and evaluation of practical static and dynamic program analyses to improve information privacy on mobile devices, with the goal of safeguarding sensitive data on user's devices [19, 26]. He is also interested in applying GPUs to new classes of workloads [65].