ABSTRACT

GRACE, MICHAEL C. Privilege Leakage in Practical Systems. (Under the direction of Xuxian Jiang and William Enck.)

The modern world places great reliance on computer systems. As the cost of processing power drops, many aspects of our daily life are now becoming “smart.” Computers used to take up a room, then an expensive piece of hardware upon a desk, and now are carried in the pockets of most people. This evolution obviously owes much to advances in hardware, but software has also played an important role: as additional layers of abstraction have been built up, it is easier than ever to develop software to do sophisticated tasks. Since computer systems today are networked, always-on, and contain valuable data, attackers are eager to use any gaps in their architecture to gain access to them. In this dissertation, we examine the ways in which privilege can be leaked through such architectural gaps in practical systems.

First, we deal with a class of memory safety issues within the Linux kernel. Most modern processor architectures use the concept of memory pages to manage memory, both to support virtual addressing and to enable page-level protection bits to describe the context in which each page should be used. Due to improper memory layouts within commodity kernels, a single page sometimes contains both executable code and writable data. An attacker can therefore use a memory write to inject their own code directly into kernel space, gaining its privilege. To address this problem, we propose using a security hypervisor to manage the kernel’s memory and enforce the \( W \oplus X \) property, which states that a given memory page can be either executable or writable, but not both at the same time. By efficiently emulating the older Harvard architecture, we can enforce a strict separation between code and data accesses even when pages contain both.

We then turn our focus to Android, a relatively new operating system that relies heavily on inter-process communication and a rich framework, where privilege is represented by a permission model. Unfortunately, such permission models can be subverted by classical “confused deputy” attacks. Such attacks can occur when a trusted party, such as an app that has a sensitive permission, performs operations on behalf of an untrusted entity. We call such vulnerabilities capability leaks, and further classify them into two broad categories. External capability leaks cross app boundaries, allowing malicious apps to either collude with or dupe trusted apps to expand their privilege; conversely, internal capability leaks leak privilege within a single app that contains components written by multiple parties. We identify two major sources of such vulnerabilities in the platform as it stands today: firmware customizations and embedded advertisement libraries. By conducting systematic surveys of these subjects, we discover several new problems and measure their prevalence, providing a solid foundation upon which to build defensive solutions or motivate architectural changes.
© Copyright 2017 by Michael C. Grace
All Rights Reserved
Privilege Leakage in Practical Systems

by

Michael C. Grace

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2017

APPROVED BY:

Ting Yu
Matthias Stallmann

Xuxian Jiang
Co-chair of Advisory Committee

William Enck
Co-chair of Advisory Committee
DEDICATION

To my mother and sister: Happy Birthday!
The author was born in Florida, moved to Connecticut as a child, and split the difference by living in North Carolina for most of his adult life. He wound up growing up alongside the IBM Personal Computer, so it was not a surprise to anyone that he would decide to pursue Computer Science as a vocation; however, settling on something a bit more specific than that took a bit of time. Having sampled (among other things) bioinformatics, artificial intelligence, grid computing, and accessibility, he finally came back to computer security, after a fortuitous meeting with his advisor-to-be, Dr. Xuxian Jiang. In August 2013, he took a position with Samsung running an enterprise security team for their mobile division. Having completed this dissertation, he intends to continue working in industry. He enjoys sponsoring related research in how to better lay out systems for security, while stumbling upon (and defusing) architectural issues similar to those described here.
ACKNOWLEDGEMENTS

This dissertation was a long time in coming, and I have many people to thank for its successful completion – so many, in fact, that I quail at the thought of trying to name them all individually, for it is nearly certain that I will forget someone vitally important. Therefore, please forgive me if I mention only a very few people by name.

To begin with, I would like to thank my co-advisors: Dr. Xuxian Jiang and Dr. William Enck. Both exhibited great patience in dealing with my meandering focus (or lack of it!) throughout great portions of my graduate career, and I learned a great many things about research, academia, and life along the way thanks to their efforts.

I would also like to thank my academic brothers and sisters: the many labmates that I worked alongside during this process. Writing this dissertation, it warmed my heart to cite the post-graduate work of many of them, and I wish them all continued success wherever they choose to apply their talents. To those that have not yet graduated: keep at it, if I can do it I am sure that you can!

I could not have completed this dissertation without the support of my colleagues at Samsung, a substantial number of whom also have work cited here. While I work in a product security capacity within a very results-oriented company, we clearly still have a strong dedication to research. I would particularly like to thank my superior, Dr. Peng Ning, for not only tolerating but encouraging my occasional absences to work on this dissertation.

The Department of Computer Science I owe a further debt of gratitude. I count several of the faculty among the most important influences in my life, and the Office of Graduate Programs was very helpful in ensuring I completed all the many arcane rituals involved in graduating. However, it is perhaps unfair to stop there: I have spent a substantial portion of my life affiliated with North Carolina State University, and it has been a very beneficial association. I had many choices to consider when choosing a university, but I feel I could not have made a better one.

Last but not least, my friends and family have always been a stabilizing influence on my life, and I am very thankful for their support.
# TABLE OF CONTENTS

## List of Tables

1. Motivation ........................................... 2
2. Thesis Statement .................................... 4
3. Contributions ....................................... 5
4. Dissertation Organization ......................... 6

## List of Figures

1. Kernel Level Privilege Separation .................. 7
2. Advances in Kernel Attacks and Defenses ........ 7
3. Kernel Rootkit Detection ............................ 8
4. Kernel Rootkit Prevention .......................... 9
5. Protecting Userspace from the Kernel ............ 10
6. TLB Manipulation ................................... 11

## Chapter 1 Introduction

1.1 Motivation ......................................... 2
1.2 Thesis Statement ................................... 4
1.3 Contributions ...................................... 5
1.4 Dissertation Organization ......................... 6

## Chapter 2 Related Work

2.1 Kernel Level Privilege Separation ................ 7
  2.1.1 Advances in Kernel Attacks and Defenses .......... 7
  2.1.2 Kernel Rootkit Detection ........................ 8
  2.1.3 Kernel Rootkit Prevention ....................... 9
  2.1.4 Protecting Userspace from the Kernel ............ 10
  2.1.5 TLB Manipulation ................................ 11

2.2 Application-Level Capability Leaks .............. 12
  2.2.1 Early Desktop Leak Detection and Defense ........ 12
  2.2.2 Early Work on Capability Leaks in Mobile Platforms . 12
  2.2.3 Static Capability Leak Analysis in Android ........ 13
  2.2.4 Static Information Leak Analysis in Mobile Platforms ... 14
  2.2.5 Framework-Level Permission System Refinements .... 15
  2.2.6 System-Wide Information Flow Monitoring on Android .. 16
  2.2.7 Application-Level Permission Policy Enforcement ..... 16
  2.2.8 Studies of Android Firmware Customization ........ 17

2.3 Sub-Application Level Capability Leaks .......... 18
  2.3.1 App Surveys .................................... 19
  2.3.2 Sub-Application-Level Capability Leak Defenses .... 20
  2.3.3 Privacy-Preserving Ad Delivery .................. 21

## Chapter 3 Kernel-Level Privilege Separation

3.1 Background ......................................... 22
3.2 Key Concepts ....................................... 25
  3.2.1 Harvard Architecture ............................ 25
  3.2.2 Virtualization and Shadow Paging ............... 26

3.3 Design .............................................. 28
  3.3.1 Page-Level Redirection for $W \oplus X$ .......... 28
  3.3.2 Mode-Sensitivity for $W \oplus KX$ ................ 31

3.4 Implementation ...................................... 32
  3.4.1 Page-Level Redirection .......................... 33
  3.4.2 Mode-Sensitivity Support ....................... 34

3.5 Evaluation .......................................... 36
<table>
<thead>
<tr>
<th>Chapter 4 Application-Level Capability Leaks</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Background</td>
<td>40</td>
</tr>
<tr>
<td>4.2 System Design</td>
<td>41</td>
</tr>
<tr>
<td>4.2.1 Explicit Capability Leak Detection</td>
<td>43</td>
</tr>
<tr>
<td>4.2.2 Implicit Capability Leak Detection</td>
<td>47</td>
</tr>
<tr>
<td>4.3 Implementation</td>
<td>48</td>
</tr>
<tr>
<td>4.3.1 Control-Flow Graph Construction</td>
<td>49</td>
</tr>
<tr>
<td>4.3.2 Capability Leak Detection</td>
<td>50</td>
</tr>
<tr>
<td>4.4 Evaluation</td>
<td>52</td>
</tr>
<tr>
<td>4.4.1 Results Overview</td>
<td>53</td>
</tr>
<tr>
<td>4.4.2 Case Studies</td>
<td>54</td>
</tr>
<tr>
<td>4.4.3 Performance Measurement</td>
<td>58</td>
</tr>
<tr>
<td>4.5Discussion</td>
<td>59</td>
</tr>
<tr>
<td>4.6 Summary</td>
<td>61</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 5 Sub-Application-Level Capability Leaks</th>
<th>62</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Background</td>
<td>62</td>
</tr>
<tr>
<td>5.2 Anatomy of an Ad-Supported App</td>
<td>64</td>
</tr>
<tr>
<td>5.3 System Design</td>
<td>67</td>
</tr>
<tr>
<td>5.3.1 Sampling Google Play</td>
<td>67</td>
</tr>
<tr>
<td>5.3.2 Analyzing Ad Libraries</td>
<td>69</td>
</tr>
<tr>
<td>5.3.3 Identifying Possible Risks</td>
<td>70</td>
</tr>
<tr>
<td>5.4 Prototyping and Evaluation</td>
<td>72</td>
</tr>
<tr>
<td>5.4.1 Overall Results</td>
<td>73</td>
</tr>
<tr>
<td>5.4.2 Categorized Findings</td>
<td>79</td>
</tr>
<tr>
<td>5.4.3 Performance Measurement</td>
<td>82</td>
</tr>
<tr>
<td>5.5 Discussion</td>
<td>82</td>
</tr>
<tr>
<td>5.6 Summary</td>
<td>84</td>
</tr>
</tbody>
</table>

| Chapter 6 Conclusion and Future Work                       | 85 |

| References                                                 | 87 |
LIST OF TABLES

Table 3.1  Effectiveness of our system ........................................... 36
Table 3.2  Software configuration for performance evaluation ..................... 37
Table 3.3  Application benchmark results. For **make**, lower is better; for Apache,
            higher is better. .......................................................... 37

Table 4.1  The list of 13 representative permissions in our study .................. 51
Table 4.2  Eight studied Android smartphones ...................................... 52
Table 4.3  Capability leak results of eight Android-based smartphones ............ 53
Table 4.4  Processing time of examined smartphone images .......................... 59

Table 5.1  Permission protection levels in Android .................................... 65
Table 5.2  The overall results from the top 50 representative ad libraries .......... 74
Table 5.3  The overall results from the remaining 50 ad libraries ................. 75
Table 5.4  Processing time of analyzed ad libraries .................................... 82
### LIST OF FIGURES

| Figure 3.1 | The Harvard architecture | 25 |
| Figure 3.2 | Guest page table vs. shadow page table | 27 |
| Figure 3.3 | Page-level mode-sensitive redirection enables an efficient implementation of the Harvard architecture on top of x86. | 29 |
| Figure 3.4 | Two shadow page tables: the user-mode page table and the kernel-mode page table share the same level-1 entries, but not top-level and level-2 entries. | 35 |
| Figure 4.1 | Detecting external capability leaks with Woodpecker | 42 |
| Figure 4.2 | A discontinuity in the control flow introduced by the Android framework. | 44 |
| Figure 4.3 | The `SEND_SMS` capability leak as a call graph. | 56 |
| Figure 5.1 | Angry Birds’ `AndroidManifest.xml` file | 64 |
| Figure 5.2 | Detecting internal capability leaks with AdRisk | 67 |
| Figure 5.3 | Popularity of the top 20 ad libraries in our study | 68 |
| Figure 5.4 | Number of ad libraries contained by each app | 76 |
| Figure 5.5 | Classes in the obfuscated `com.mt.airad` package | 77 |
| Figure 5.6 | Plankton handshake communication | 81 |
Chapter 1

Introduction

The modern world places great reliance on computer systems. As the cost of processing power drops, many aspects of our daily life are now becoming “smart.” Computers used to take up a room, then an expensive piece of hardware upon a desk, and now are carried in the pockets of most people. This evolution obviously owes much to advances in hardware, but software has also played an important role: as additional layers of abstraction have been built up, it is easier than ever to develop software to do sophisticated tasks.

For example, consider the smartphone. This breed of sensor-studded device is no more inconvenient to carry around than a feature phone, but offers far greater functionality, being limited only by the creativity of the user base. Adoption was rapid: the launch of the iPhone in 2007, followed closely by the first Android device a little over a year later, kicked off an era of explosive growth in the “smartphone” segment. In 2006, smartphone sales represented 64 million units [8] out of a 1 billion strong mobile phone market [17]; ten years later, smartphones now represent the lion’s share of mobile phone sales, having shipped 1.5 billion units in 2016 [6, 9].

From another perspective, two new vendors entered an existing (if small) segment, and ten years later had displaced all other platform vendors: devices running Apple’s iOS and Google’s Android now account for 99.6% of all smartphone sales, with Android claiming 81.7% alone [16].

To get to this level of market dominance, both of these platforms traded heavily on their strategy of embracing developers. Developing and distributing apps on these smartphone platforms is very easy: they offer powerful, standardized frameworks alongside easy-to-navigate app markets and user-friendly installation and uninstallation mechanisms. Previous smartphone operating systems tended to rely on a core of professional developers to produce apps, employed by the companies that designed, made and sold the device (i.e., the platform vendor, the manufacturer, and the carrier). Android went a step further, combining commodity software components with an open-source stance that allowed manufacturers, carriers, and users to add functionality to the core operating system itself (although Google still curates this baseline).
Another example presents itself in the choice of operating system kernel adopted by the Android project: Linux. Released in 1991 as a non-commercial, free open-source effort, its low cost and ease of modification ultimately led to grass-roots popularity with IT professionals. By 1999, over half of web domains used Linux [31], and by the present day that number is now approximately two thirds [29]. The tunable nature of the operating system also led to broad adoption in the field of high-performance computing: 291 of the top 500 fastest supercomputers by 2006 – merely six years after first appearing on the list – ran a variant of Linux, a number that has climbed to 498 today [23]. Due to its free nature, it is harder to accurately estimate the amount of backend business systems running Linux today, but the platform has clearly seen broad adoption in such roles across the world.

These two examples showcase the speed with which disruptive innovation can occur in software – even for software as fundamental as an operating system platform. Furthermore, the rapid adoption of a new paradigm in both these contexts was driven by buy-in from many different parties to collaboratively improve the software, to allow it to do new things and colonize new use cases. Given the number of architectural layers in such complex systems, however, it is natural to overlook implicit security assumptions when trying to cover new ground, or to have difficulty addressing new threat models using the existing architectural layout.

1.1 Motivation

Since computer systems today are networked, always-on, and contain valuable data, attackers are eager to use any gaps in their architecture to gain access to them. In the modern world, such access can be traded for financial or even ideological gain, by stealing data, setting up botnets to attack internet resources, or through simple vandalism.

To return to the previous examples, since the Linux kernel is used so broadly, vulnerabilities that effect it are big news. The recent “Dirty COW” flaw [13] attracted a lot of media attention because the vulnerability had gone undetected in the kernel for nine years, and was first brought to light by what appeared to be an active, in-the-wild exploitation campaign. Given the wide spread of the Linux kernel, and the data stored on so many network-connected information systems, the potential for damage was great – practically every Linux machine running at the time of discovery was theoretically vulnerable.

Dirty COW is an example of a memory safety issue, where a security vulnerability arose from a memory access. A mistake in the Linux’s memory-management subsystem resulted in a situation where an attacker could turn read-only file mappings into writable ones, enabling them to interfere with the input to more trusted components. Leveraging this exploit primitive to gain privileged access is not difficult; for example, within four days, a public, working proof-of-concept exploit had been developed that would achieve root on Android smartphones [11].
Memory safety issues account for a huge proportion of flaws within privileged components. Another memory safety vulnerability, stagefright [83], inspired similarly impressive headlines as media outlets claimed hundreds of millions of devices might be impacted. Stagefright was simply a series of mistakes within the native code that Android used to render media files. This code, executing at privilege, could be commandeered by an attacker who tricked the phone into rendering a malformed media file. While the issue was addressed swiftly by the Android community, Google’s retrospective about these events contained the worrying statistic that 85.9% of all the security-relevant bugs within the platform involved memory safety [117].

Google attempts to mitigate memory safety issues in Android by encouraging the use of a memory- and type-safe language, Java. Java is easy to develop in, especially with Google’s strong investment in rich tooling and automation; this allows users to easily extend and supplement the built-in functionality that ships with their smartphones. As a result, the Google Play Store currently allows users to choose from over 2.8 million apps [22]. Unsurprisingly, everyday people are increasingly relying on smartphones to manage their personal data. Inside the phone, we can find current (or past) geo-location information about the user [19], logs of placed and received calls, an address book with rich contact information, as well as cached emails and photos taken with the built-in camera. The type and the volume of information kept in the phone naturally leads to concerns [86, 87, 128, 194] about the safety of this private information, including the way it is managed and accessed.

To mediate access to personal information and certain sensitive phone functions, smartphone platform vendors have explored a number of approaches. For example, Apple uses a vetting process through which each third-party app must be scrutinized before it will be made available in the app store. After installing an app, Apple’s iOS platform will prompt the user to approve the use of some functions at run-time, upon their first access. Analogously, Google defines a permission-based security model in Android by requiring each app to explicitly request permissions up-front to access personal information and phone features.

Unfortunately, such permission models can be subverted by classical “confused deputy” attacks [104]. Such attacks can occur when a trusted party, such as an app that has a sensitive permission, performs operations on behalf of an untrusted entity. We call such vulnerabilities capability leaks, and further classify them into two broad categories. External capability leaks cross app boundaries, allowing malicious apps to either collude with or dupe trusted apps to expand their privilege; conversely, internal capability leaks leak privilege within a single app that contains components written by multiple parties.
1.2 Thesis Statement

While they appear to be quite different issues, memory safety and capability leaks can be seen as simply the same general problem at a different level of granularity. Memory safety issues come from a system architecture that does not properly take into account the intent of a memory access, which allows an attacker to “trick” the system into corrupting its state. On the other hand, capability leaks involve an incomplete protection boundary around a privilege; for an external capability leak in Android, this is usually a missing permission check, while internal capability leaks involve a disconnect between the trust model of the permission system and the process boundary granularity at which it is enforced.

Summarizing this more succinctly, we derive the central thesis of this work:

Practical systems contain subjects at all levels (e.g., kernel, application, and sub-application) that reexpose privileged access to information and resources due to improper architectural organizations that are not dependent on functional requirements.

Viewed through this lens, both memory safety issues and capability leaks are part of the broader class of privilege leaks. In this dissertation, we explore these privilege leaks at each level of abstraction within a typical system, ultimately focusing on Android as an interesting example due to its focus on the framework and inter-process communication.

First, we deal with a class of memory safety issues within the Linux kernel. Most modern processor architectures use the concept of memory pages to manage memory, both to support virtual addressing and to enable page-level protection bits to describe the context in which each page should be used. Our work focuses on the x86 architecture, where the minimum page size is 4 kilobytes; this granularity means that sometimes memory with different intended purposes is allocated on the same page, which therefore over-privileges that page. When applied to an operating system kernel, this can result in a context where a page is mapped both writable and executable at the same time, allowing a memory safety problem to be used to inject code into kernel space. To address this problem, we propose using a security hypervisor to manage the kernel’s memory and enforce the \( W \oplus X \) property, which states that a given memory page can be either executable or writable, but not both at the same time. While operating system kernels now strive to conform to this property, the legacy architecture of Linux did not initially consider \( W \oplus X \). Fortunately, by using the underlying hardware in an unanticipated way, we can address this problem by transparently splitting every page that is both writable and addressable into two physical pages. Furthermore, without having hardware support for this concept, we are able to use the same hypervisor to ensure that user-space code never runs at kernel privilege, blocking another avenue of code injection.
We then turn our focus to a survey the landscape of a representative smartphone platform, Android, by developing tools and techniques to identify potential capability leaks. We identify two major sources of such vulnerabilities in the platform as it stands today: firmware customizations, which lead to external capability leaks, and embedded advertisement libraries, which correspondingly constitute internal capability leaks. The Android platform does not offer the user much recourse in addressing either problem; for example, many phones are sold in a “locked” configuration that prevents the user from modifying the firmware, yet carriers and manufacturers can be slow to port newer versions of Android to existing phones. Similarly, ad-supported apps rarely disclose what advertisement libraries they contain, or how those libraries use the access the user has granted to the app.

1.3 Contributions

This dissertation makes the following contributions:

- **We propose a hardware virtualization-based Harvard architecture, hvmHarvard [98], to effectively protect commodity OS kernels from kernel code-injection attacks.** Compared with contemporary approaches, our system achieves the most robust protection guarantees with far improved performance (under < 5% overhead), without the need to modify the kernel.

- **We perform the first systematic evaluation of the impact of vendor customizations on the security and privacy of Android smartphones [100].** Using a static analysis system, Woodpecker, we document and overcome several challenges in resolving the possible paths execution within an Android app can take. Our results indicated that of the thirteen privileged permissions we studied, eleven were leaked by at least one of the phones we tested. These findings were severe enough to be referenced in a Federal Trade Commission (FTC) action against one manufacturer [18].

- **We conduct the first large-scale survey of in-app ad libraries in Android [99], uncovering several serious privacy and security risks.** Having collected a sample of 100,000 apps, we identify 100 ad libraries, characterize their behavior, and measure their prevalence. We observe several ad libraries engaging in risky behaviors: making opportunistic use of the permissions of the host app without the knowledge or consent of either the developer or the user, permissively allowing advertisers to directly exfiltrate data about the user, and even downloading and executing unauthenticated code within the context of the host app.
1.4 Dissertation Organization

The remainder of this dissertation is organized as follows. First, to set the work in context, Chapter 2 surveys closely related work. The following chapters then address the core issue of improper privilege separation at ever-finer levels of granularity. To start with, Chapter 3 shows how memory safety issues within the Linux kernel are an artifact of the architecture modern processors use to relate to memory – and solves them by looking to older systems for inspiration. Chapter 4 studies the role of vendor customizations in the framework-centric Android ecosystem, and illustrates how a new concept of an operating system architecture can lead to security issues. Chapter 5 presents a large-scale survey of Android apps themselves, showing how assumptions about the trust model implicit in Android can be violated by the common practice of including third-party libraries. Lastly, Chapter 6 concludes this dissertation and presents some avenues of future work.
Chapter 2

Related Work

2.1 Kernel Level Privilege Separation

Due to the value inherent in holding the figurative “high ground,” i.e., the greatest privilege, within a system, there is a ceaseless arms race between exploit writers and those attempting to architect secure systems. When we put forward hvmHarvard, code injection attacks were a major vector. Kernels have since largely adopted $W \oplus X$, and thus attack techniques have evolved. While it is difficult to deny that attack techniques are based around misusing existing architectural mechanisms, it seems less obvious that many defensive techniques are implemented by repurposing existing mechanisms in an unanticipated way.

2.1.1 Advances in Kernel Attacks and Defenses

As stated previously, since hvmHarvard was put forward, kernels have largely found ways to conform to $W \oplus X$; indeed, at the same time hvmHarvard was being written, Liakh et al. [120] were finding ways to make the Linux kernel conform to the property.

It took longer to address so called “ret2usr” attacks. In such attacks, the kernel is tricked into accessing userspace memory in kernel mode. hvmHarvard additionally enforced a $W \oplus KX$ property, ensuring that the kernel could only execute its own code. However, eventually hardware support arrived; modern processors and kernels limit the kernels capability to access userspace memory via two bits (variously called either SMAP and SMEP [71] or PAN and PXN [41], depending on architecture). These mirror the much-older “supervisor” bit, in that when the processor is executing in supervisory (i.e., kernel) mode, either reads (for SMAP/PAN) or instruction fetches (for SMEP/PXN) will fault.

Unfortunately, these new mitigations again depend on the kernel conforming to the proper architectural layout. The “ret2dir” attack [114] illustrates that again architectures cannot be so easily changed, which leads to an exploitable gap. The Linux kernel maps every physical
frame in a direct mapping of physical RAM (i.e., there is an identity map of all memory in the kernel’s virtual address space). It also maps physical frames into user processes, creating a double map “synonym” of the same physical frame in userspace. A memory corruption kernel exploit could use this synonym address to address userspace data through kernel mappings, overcoming protections such as SMAP, SMEP, PXN, and PAN. ret2dir can be thought of as a confused deputy attack, as the kernel is tricked to accessing userspace memory through kernel mappings. To solve this issue, Vasileios et al. propose enforcing exclusive ownership of pages (either userspace or kernel) by unmapping a page from Linux kernel direct mapping when assigning to userspace, removing capability of kernel to access it using its own mapping.

However, code-reuse attacks [53, 80, 154] still enable attackers to execute code at elevated privilege. These attacks stitch together existing code at either function or sub-function granularity to implement new, and arbitrary, functionality. The attacks work because the architecture has no inherent understanding of functional units larger than an instruction. Various defenses have been proposed to force execution to only take intended paths, but one line of research in particular fits with the theme of this dissertation: execute-only memory, which can be thought of as enforcing a $R \oplus X$ property over a memory region. By removing the capability of a piece of code to both read and execute a page, it prevents derandomization of code in address space; coupled with fine-grained randomization techniques, this makes executing unintended control flows unlikely to succeed due to the inability to predict where useful instructions will be placed. Different projects achieve this in different ways: XnR [47] uses a heuristic, disabling executing after a threshold of reads; Readactor [73] uses compiler extensions to emit execute-only memory; Heisenbyte [166] and NEAR [178] are able to disallow execute after a single read, and NORAX [184] takes a binary instrumentation approach. CaSE [188] (Cache-Assisted Secure Execution) is a particularly interesting example, as it attempts to achieve code confidentiality on ARM processors by executing it within secure cache; the main memory never contains either the plaintext code to execute nor the key used to decrypt it, despite ARM not having explicit support for such a feature.

### 2.1.2 Kernel Rootkit Detection

A number of systems have been proposed to detect the presence of kernel rootkits. Some of them passively validate kernel code and examine kernel data for signs of infection. For example, System Virginity Verifier [150] validates the integrity of the Windows instance that it runs within. As running inside a compromised operating system is dangerous, Copilot [139] copies operating system memory onto a PCI card for analysis by a dedicated co-processor. Further extensions allow it to detect breaches of kernel data semantic integrity [112] and state-based control flow integrity [140]. Strider GhostBuster [173] and VMwatcher [111] aim to look for
discrepancies between an internal and external view of a system to detect the hiding behavior from rootkits. Taking a different approach, Lares [137] and its in-VM equivalent, SIM [155], attempt to create secure kernel hooks that can be used to monitor system events. In particular, SIM is capable of installing hooks into a virtualized guest that run code safely without hypervisor intervention. SIM uses the same Intel CR3 Target Value List feature that hvmHarvard does, but uses it to create a safe introspection environment instead of a new paging feature as in our system. HIMA [43] is an early ARM-based work, which uses the ARM TrustZone’s isolation from the “normal world” (i.e., the regular operating system) to place an introspection hypervisor. By preventing the kernel from performing some necessary management functions without HIMA’s direct involvement, the system can ensure it is scheduled periodically, limiting the amount of time that a rootkit can persist without being detected.

2.1.3 Kernel Rootkit Prevention

To move beyond detection, various researchers have proposed using a secure monitor to enforce properties on the operating system kernel. These are interesting in the context of our overall theme because these systems often wind up using features of the hardware architecture in an unanticipated way.

Livewire [96] is among the first in using virtualization techniques for this purpose, though the system mainly focuses on the protection of static kernel code and data structures. Secure Virtual Architecture (SVA) [75] takes a more radical approach, instead defining a virtual, low-level, typed instruction set suitable for executing all code on a system, including kernel and application code. SVA code is translated for execution by a virtual machine transparently, offline or online. Unfortunately, such a scheme incurs a fairly high performance overhead.

SecVisor [153] is a small security hypervisor that aims to securely enforce a $W \oplus X$ guarantee over memory, but it requires modifying the OS kernel for the support. Also note that SecVisor implemented a similar KX paging mode to our own work, but its shadow page table implementation uses a single page table per process, which leads to considerable performance overhead. Instead, the approach we took with hvmHarvard proposes two page tables. Further, with the CR3 Target Value List hardware virtualization feature, our system allows a guest running under our system to switch between these two page tables without hypervisor intervention.

In the same vein, NICKLE [146] aims to protect the integrity of the kernel code with a software-based implementation of the Harvard architecture. The software implementation is based on instruction-level redirection, which has a high performance overhead. In comparison, hvmHarvard proposes a page-level, mode-sensitive redirection that substantially reduces the performance overhead.
Fides [163] introduces the concept of Self-Protecting Modules, which allows isolating and protecting arbitrary modules of code from one another by using the hypervisor to place them in separate memory spaces. It is important to note that this system works within both kernel and user space. Monolithic kernels definitionally suffer from implicit capability leaks, as often opaque, proprietary driver modules are loaded with the same privilege as all other kernel code; in this light, Fides seeks to stop such leaks without rearchitecting the entire kernel.

hvmHarvard uses Xen’s Shadow Paging mechanism in order to stay abreast of events in kernel space. Shadow Paging is an expensive and complicated mechanism, as it involves the hypervisor having to interpose on kernel page table updates in order to align them with its own view of memory. Accordingly, hardware virtualization extensions have been introduced to reduce this overhead. Wang et al. propose a security virtualization framework called SecPod [172] which uses these new extensions alongside a lightweight version of classical Shadow Paging in order to provide strong guarantees with much lower overhead.

Note that while the systems we have covered so far in this section have all been based upon the Intel and AMD architecture, the problems they are attempting to solve are not exclusive to those architectures. For example, only very recently have ARM processors developed robust hardware virtualization extensions. Being unable to place a monitor in a hypervisor, researchers instead turned to the ARM TrustZone, which to that point had primarily been used for Data Rights Management software. Two similar and contemporary systems, Realtime Kernel Protection [44] and Sprobes [97], implement paravirtualized security hypervisors using the TrustZone. In order to ensure that they can enforce guarantees on the kernel, without being able to trap events, these systems attempt to select a minimum set of invariants necessary in order to achieve good performance.

2.1.4 Protecting Userspace from the Kernel

Another set of works attempt to reinforce the boundary between the kernel and userspace by reducing the amount of trust placed in the kernel itself. For an early example, Proxos [165] uses virtualization to allow applications to route system calls to a private kernel of their choice.

Another virtualization-based approach, Overshadow [64], goes rather farther: its basic premise is that the kernel cannot be trusted with sensitive user data, even if it is not compromised or actively malicious. As in hvmHarvard, Overshadow captures the mode switches to alter the view of memory inside a protected VM. However, the differences are twofold: first, hvmHarvard switches between user and kernel page tables on each mode switch but does not attempt to encrypt user memory pages. In comparison, Overshadow makes the user memory appear encrypted to the operating system kernel, yet act as normal when at user privilege. Second, the
goal of hvmHarvard is to protect the kernel from malicious user applications while Overshadow does the exact reverse.

Unfortunately, Overshadow’s use of encryption can make it difficult to share information within the operating system itself. A set of more recent systems attempt to achieve more flexible isolation while simultaneously avoiding the performance overhead of encryption operations. For example, InkTag [106] attempts to instead verify the operating system as it operates on memory pages that are marked as sensitive. Virtual Ghost [74] instead uses compiler techniques to enforce control flow integrity guarantees to protect arbitrary “ghost memory” regions from the kernel, rather than virtualization and encryption; its goal is to make accesses to such memory inexpressible even without the benefit of a hypervisor. AppShield [65] uses hardware virtualization support, specifically Nested Page Tables/Extended Page Tables support, to isolate an applications address space such that all accesses from the kernel are blocked except those explicitly authorized by the application through system calls. OSP [68] attempts to achieve similar isolation of non-TrustZone, userspace applications on the ARM platform, protecting critical code sections from compromised kernels.

All of these systems are attempting to change one of the fundamental assumptions made by many commodity architectures and operating systems, which is that an operating system kernel must be able to access the memory and control the execution of userspace processes that it manages. In effect, the kernel itself, by virtue of its position, is the source of an implicit capability leak within all the userspace applications that it runs. Recent hardware support such as Intel SGX [69] is making this at least an architectural reality, though it remains to be seen how software will ultimately adapt.

2.1.5 TLB Manipulation

Finally, the presence of separate TLBs has been recognized and exploited in other contexts for different applications. For example, Wurster et al. [180] proposes using different ITLB and DTLB mappings to attack self-checksumming code. Almost simultaneously, Sparks and Butler demonstrated a rootkit prototype called Shadow Walker [160] that could elude existing detection using the de-synchronized TLB. Later, Rosenblum et al. [148] put forward a system that used a modified version of Xen to instrument a tamper-resistant process within a VM. While the version of Xen used is unclear, it appears that their system operated on para-virtualized guests. In contrast, our system is mainly concerned with fully-virtualized guests and aims to defeat existing kernel rootkits. To the best of our knowledge, no other system has exploited recent hardware virtualization features to efficiently implement the Harvard architecture on x86, including the use of tagged TLBs to manipulate the TLBs of a guest from outside as well as the unique hardware feature of the CR3 Target Value List.
2.2 Application-Level Capability Leaks

As discussed in Chapter 1, capability leaks are a generalization of the confused deputy attack [104]. While the two surveys within this dissertation focus on the problem of capability leaks within Android, the problem is certainly not unique to the Android platform. For example, in 1999, Smith found that PC manufacturers bundled vulnerable ActiveX controls in their custom Windows installations [159], in an eerie echo of our own work. Similarly, our follow-on work with IoT systems shows them to be vulnerable to such attacks, being made up of multiple small systems communicating with one another with protocols that are unable to encapsulate the origin of a command at a sub-device level [79].

2.2.1 Early Desktop Leak Detection and Defense

To briefly summarize the some of the approaches that had been taken on other platforms before the rise of mobility, we look back at the desktop computer. A number of systems that target desktop apps have been developed to detect system-wide information flow or confine untrusted app behavior. For example, TightLip [185] treats a target process as a black box. When the target process accesses sensitive data, TightLip instantiates a sandboxed copy, gives fuzzed data to the sandboxed copy and runs the copy in parallel with the target for output comparison and leak detection. Privacy Oracle [113] applies a differential testing technique to detect the correlation or likely leaks between input perturbations and output perturbations of the application. Given the lack of clear application-level permissions to gate and identify sensitive accesses to information, these systems have to overcome additional challenges in order to determine the sensitivity of data; on the recent crop of mobile operating systems, this is a far easier problem to solve. Taking a different tack, system-level approaches such as Asbestos [170], HiStar [186], Flume [118], Process Coloring [110], and PRECIP [171] instantiate information flow at the process level by labeling running processes and propagating those labels based on process behavior. Again, due to the lack of robust permission systems within these desktop operating systems, this body of work attempts to label processes as sources of sensitive data, rather than APIs, a loss of some granularity; however, similar information-flow-based approaches have been applied to mobile operating systems, taking advantage of the additional semantics available there.

2.2.2 Early Work on Capability Leaks in Mobile Platforms

From very early after Android’s release, researchers were probing its security model to look for vulnerabilities [77, 82, 135]. For example, Davi et al. [77] show a manually-constructed confused deputy attack against the Android Scripting Environment. QUIRE [82] anticipated such
attacks; it proposes a means to allow apps to reason about the call-chain and data provenance of requests, which could be potentially helpful in mitigating such confused-deputy attacks. Nils [135] manually analyzed the HTC Legend’s system image looking for possible permission abuses, and uncovered several. In comparison to these previous efforts, the two surveys we will present both aim to systematically detect such gaps in the security model, enabling the problem to be studied at a greater scale.

Meanwhile, note that some Android malware such as Soundcomber [151] were developed by requesting certain Android permissions. Our results from Woodpecker show that these requests could be potentially avoided as the permissions might have already been leaked (e.g., as in the case of \texttt{RECORD\_AUDIO}).

Felt et al. [93] proposed the notion of permission re-delegation in the a generalized context applicable for both web and smartphone apps; we view the problem of permission re-delegation as a form of explicit capability leak, but distinct from implicit capability leaks (which do not make use of any public interface for permission inheritance).

Another related work, Stowaway [91], is designed to detect overprivilege in Android apps, where an app requests more permissions than it needs to function. The problem of overprivilege is certainly related to the kinds of issues we discover in our works: in AdRisk, we find that privileges not being used by the host app may be opportunistically leveraged by the libraries that it contains, while Woodpecker can instead be thought of as focusing on underprivilege in Android apps, where an action that should normally require a declared capability can be performed without requesting it.

### 2.2.3 Static Capability Leak Analysis in Android

Other systems have sought vulnerabilities in inter-process communication within and across apps using static techniques, similar to those employed by Woodpecker. ComDroid [66] looked for a variety of problems related to improperly permission-protected inter-process communication on Android, looking at the top 100 most popular apps on the Android Market. This did uncover 13 cases in which activities could be improperly triggered by unauthorized third parties, but did not uncover any of the more serious issues Woodpecker was able to, presumably due a combination of the more-privileged nature of preloaded applications and the larger scope of our study. DroidChecker [62] performed a larger study of 1179 apps, but though the system only detected 6 true-positive external capability leaks. Meanwhile, chex [127] had a very similar analysis architecture to Woodpecker, but again focused on colluding third-party apps rather than dealing with leaks stemming from framework customization efforts; however, interestingly, it was capable of stitching together flaws across components to some extent, something Woodpecker left entirely to a human analyst. COVERT [48] was designed to detect cross-app capability
leaks by combining static analysis with a model checking approach; their evaluation discovered several combinations of apps that could be combined to actually (inadvertently) exploit such leaks.

ContentScope [192], on the other hand, studied a component type that other systems did not handle: ContentProviders. This study of 62,519 apps discovered that approximately 2.3% of them did not protect access to their internal databases, which either could lead to an indirect leak of sensitive data or to content-pollution attacks that could cause vulnerabilities within the app. DBDroidScanner [105] goes farther, attempting to additionally identify private database accesses – changes to internal app state via unprotected external interfaces other than the intended ContentProvider interface.

Finally, two systems deserve special mention, as they directly attempt to defend against external capability leaks. The first, AppSealer [187], uses static taint analysis to automatically patch apps that contain such component-hijacking vulnerabilities. The second, RoppDroid [76], uses resource virtualization to prevent privilege escalations while preventing crashes; instead of denying access to a resource that is considered too privileged to be returned to a given caller, a dummy resource is constructed for this purpose. This scheme can be thought of as similar to our work on hvmHarvard, as it too redirects accesses transparently based on their context.

2.2.4 Static Information Leak Analysis in Mobile Platforms

There have been a few works that have aimed to analyze mobile apps for information disclosure problems. One of the earliest, PiOS [86], is a representative example, which constructs a control-flow graph for an iOS app and then looks for the presence of information-leaking execution through that graph. Specifically, PiOS tries to link sources of private information to network interfaces, as iOS offered very few avenues for inter-app communication at the time. FlowDroid [42] aims to statically perform context, flow, field, object-sensitive and lifecycle-aware taint analysis on Android apps, in order to be able to detect information flow problems without having to run an app; the challenges the authors faced led to a follow-on work, EdgeMiner [59], to attempt to more completely summarize the flow of information through the Android framework. These systems relate to a larger body of work, discussed later in this section, that focuses on information flow rather than permission policy; however, despite the difference in goals, the techniques used and challenges faced have much in common with these other works. For example, Woodpecker relies on summaries of framework callbacks, as well; incorporating the output of a system like EdgeMiner [59] might be able to improve the fidelity of its analysis.
2.2.5 Framework-Level Permission System Refinements

As alluded to in our discussion of desktop operating system works, the permission system itself, where an app contains a manifest that describes its capabilities, is a concept just recently coming into vogue in commodity OSes: for example, the same year that Android was released, Dragoni et al. proposed “Security-by-Contract” [81] to secure .NET mobile devices, arguing that code should not only be signed to validate the identity of the publisher, but should also have semantics about what it is supposed to do attached to it. However, deciding the proper amount of semantic information to include, as well as the way in which to use that information to protect the user, is non-trivial. Android’s initial design was to simply present the user with a list of all permissions requested by an app, highlighting the most dangerous ones, and ask for them to grant them all in order to install an app. To defend against both capability and information leaks in Android, a number of extensions to this permission system have been proposed.

For example, work by Chaudhuri et al. [63, 95] formalizes data flow on Android so that a data flow policy can be formally specified for an Android app, which can then be checked against the app code to ensure compliance. A SCanDroid system [95] has been accordingly developed to extract such specifications from the manifest file that accompanies such applications, and check whether data flows through the app are consistent with the specification. Note that SCanDroid requires accessing the app’s Java source code for the analysis, a difficult-to-meet requirement that many later systems are able to do without. Kirin [89] checks the manifest of apps that are being installed against a permission-assignment policy, blocking any that request certain potentially unsafe combinations. Saint [136] takes this approach a step further, by allowing app developers to constrain permission assignment at install-time and permission use at run-time. These systems have the disadvantage of either requiring app developer involvement or potentially blocking the installation of safe (if possibly overprivileged) apps, drawbacks which the another line of research attempted to resolve by looking at permission use and assignment more directly.

For example, MockDroid [50] allows privacy-sensitive calls to be rewritten to return “failure” results. Similarly, TISSA [194] protects user information by modifying the Android framework to support user-defined information disclosure policies; sensitive APIs can return false information under such a scheme instead of simply failing. AppFence [107] further refines this approach by adding taint-tracking, allowing yet more nuanced policies. Other systems try to add further expressivity to the permission system, such as Apex [134], which modifies the framework to allow permissions to be selectively granted and revoked at run-time.

Android ultimately incorporated a coarse-grained version of this last operating-system-level approach in Android 6.0 “Marshmallow” [10]; while only permitting the user to control the
dangerous" permissions of an app, and then only at a group level, the new “Dynamic Permissions” model does provide better visibility into the circumstances in which an app uses a permission compared to the previous “all or nothing” install-time model.

2.2.6 System-Wide Information Flow Monitoring on Android

As mentioned earlier, on desktop operating systems there had been some work on attempting to secure sensitive data by adopting policies that track data, rather than that gate access to APIs that expose that data. The line is a bit blurry; for example, XManDroid [55], has a similar intuition to QUIRE [82] – that provenance could be a useful additional dimension to the Android permission system – but goes a step further, proposing a policy enforcement mechanism built atop the Android framework with the express purpose of curbing confused-deputy attacks. This system tracks the provenance of a request, rather than tracking the data it contains. Bugiel et al. [56] later extend the same framework into the kernel; this line of research ultimately connected [58] to efforts to bring SELinux to Android [158], though sadly the commercialized version of SE for Android lacks these functions today.

However, other systems took a more pure approach to information flow tracking and policy management. TaintDroid [87] applies dynamic taint analysis to monitor information-stealing Android apps. Specifically, by explicitly modeling the flow of sensitive information through Android, TaintDroid raises alerts when any private data is going to be transmitted from the device. Other works have run the full gamut of dynamic analysis methods since: for example, DroidScope [182] similarly collects traces and tracks information flow using taint, but does it through virtualization; DroidInjector [90] has similar aims, but uses process injection through the standard ptrace mechanisms.

Aquifer [133] can be viewed as a direct descendant of some of the desktop works [170, 186, 118], as it customizes the Android framework to support decentralized information flow control (DIFC); apps can define export restrictions on their data, while still allowing intermediaries that form part of the same UI workflow to operate on that data. Weir [132] adds support for intermediaries that are background components, as well as the ability to define declassifiers, as in Flume [118].

2.2.7 Application-Level Permission Policy Enforcement

There has been a huge amount of interest in the research community about how best to enforce policies on the behavior of apps on Android, without rooting or modifying the underlying operating system [45, 46, 109, 144, 181]. All these systems attempt to allow policy enforcement without modifying the framework, instead rewriting apps. The key differences between them principally involve how they manage their policy. Systems like Dr. Android and Mr. Hide [109]
deprivilege the original app, forcing requests from it to travel through a policy-enforcement app. Meanwhile, Aurasium [181] and AppGuard [46] embed the monitoring code within the rewritten app, which retains privileges but has additional policy enforcement checks; these systems are robust so long as the checks are properly inserted, which can be a challenge given the difficulty in modeling control flow within Android apps. DROIDFORCE [144] can be thought of as an evolution of those previous two systems, as its inline monitoring code is injected based in part on the output of a static data flow analysis engine, and its policy decisions are managed by an external app across an entire system – which may permit blocking some libraries from exfiltrating private data, but might also block legitimate app functionality. Boxify [45] can be viewed as an extension of the authors’ previous work on AppGuard [46], using the Isolated Process (previously intended for Web browsers to isolate their JavaScript engines) to put the reference monitor out of context, preventing tampering with it and ensuring items not caught by the rewriting step are unable to act with privilege.

AppCage [193] and NJAS [51] are a bit different, as these systems that do not require altering the code of the app to be sandboxed; instead, that app runs within the context of another app, and system call interposition is used to ensure all inter-process communication events (among other privileged accesses) are caught by the monitor – including ones that originate from native code. In 2016, a follow-on survey from the group that produced NJAS [38] was used to gain insights into valid and invalid use of native code; from this, the authors derived a policy that could be used to enforce stricter sandboxing of native code while allowing 99.77% of legitimate apps to still run properly. As impressive as this body of work is, these systems treat the apps as a whole, without further differentiating any embedded libraries from the host app’s code; it is possible to use them to revoke permissions from apps that do not need them, but this is a trial-and-error process that may impair an app’s legitimate functionality.

### 2.2.8 Studies of Android Firmware Customization

Once our findings from Woodpecker were published, several follow-on works aimed to judge the true extent of the problem. One of the first was our own refinement of the techniques used by Woodpecker, called SEFA [179]. Like chex [127] and COVERT [48], it added the ability to study cross-component and cross-app vulnerabilities, but like Woodpecker, it focused on vendor customizations of firmware. The results showed that approximately 8.9% of the vulnerabilities within firmware involved multiple framework apps operating in conjunction. SEFA also was a longitudinal study, comparing two different generations of smartphones from each of five vendors; the results were disheartening, as the incidence of vulnerabilities appeared to be largely stable over time.
IntentFuzzer [183], in contrast to previous work in this area, used a dynamic analysis technique, by fuzzing the exposed interfaces of apps from both the Google Play store as well as the firmware of two smartphones. Their results indicated an approximately 5.6% prevalence of external capability leaks in apps from the Google Play store, compared to an even 10% among apps from the firmware that they tested.

Taking a different approach from these other works, DexDiff [130] evaluated firmware images statically, but by using graph isomorphism techniques to identify differences between a manufacturer’s customized firmware and the corresponding Google baseline. This approach was not directly aimed at discovering capability leaks, but proved useful in detecting customizations that might have negative security and privacy; for example, DexDiff handily isolates the hooks that CarrierIQ [85] injects into phones for its controversial carrier-analytics functionality.

The last class of works surveyed firmware images, but looked for different kinds of capability leaks than what we sought with Woodpecker. ADDICTED [191] sought vulnerabilities in device drivers in four Android devices using dynamic analysis techniques, then performed a large-scale differential-analysis study of 2423 firmware images from around the world. The results indicated that privileges were being assigned incorrectly in many cases, for example allowing unprivileged apps to directly drive the camera or capture the contents of the screen. DroidRay [190] looked at another interesting class of firmware problems, namely maliciously customized firmware, looking for malware contained within firmware images using traditional techniques. Systems like Woodpecker and SEFA focused on major manufacturers’ official firmware, which can be assumed to lack malware due to the risk of reputational damage and the resources these companies can bring to bear; smaller makers may not enjoy such resources, and of course some customized firmware may have been put together with malicious intent. This study did find several issues, but neglected to look for any form of collusion attack or capability leaks; it appears that the issue of capability leaks in smaller OEMs and custom firmware remains incompletely studied today, despite the growth of Android in developing markets [15]. Finally, Harehunter [36] looked for “hanging attribute references” in vendor-customized firmware; these flaws constitute another form of external capability leak, one in which the privileges are granted to a subject by its name rather than the permissions that it holds. For example, we discovered that some tablet devices, not having phone functionality, lacked the Android dialer app; however, other privileged software expected it to be present, and would grant additional access to any package having the same name.

2.3 Sub-Application Level Capability Leaks

While most of the work relevant to the Woodpecker system is also relevant to AdRisk, we will avoid covering works that are more tangentially related here. Instead, this section focuses on
those efforts that directly touch on the issue of third-party code being included within the context of a host app, as well as the general problem of advertisements.

2.3.1 App Surveys

Enck et al. [88] wrote the Dalvik decompiler to study around one thousand popular Android apps, and reported a number of findings about them, including the presence and impact of ad libraries. AdRisk studied one hundred thousand apps, which allowed us to systematically identify and assess a wider variety of ad libraries. For example, none of the libraries we discovered that feature dynamic code loading were included or reported earlier.

Another early study by Felt et al. [91] aimed to identify the amount of “overprivilege” in Android apps. (An overprivilege occurs when an app requests more permissions than it uses.) In particular, among 940 Android apps being studied, more than one third were found to be overprivileged. Given the permission-probing behavior we detected in existing ad libraries, it is possible that even more apps are requesting unnecessary permissions, which are then opportunistically being used by their embedded ad libraries. Dynamic code loading paints a yet more grim picture, as we found one ad library uploaded the permissions its host app was granted before downloading the code. Poeplau et al. [141] later looked exclusively for dynamic code loading in a sample of 1,632 popular apps, and discovered that 9.25% load external code insecurely – as do 16% of the top 50 free applications. They argue that the problem is severe enough to warrant a mandatory verification mechanism to check code integrity before it is dynamically loaded.

Another survey by Chin and Wagner [67] analyzed WebView vulnerabilities in a survey of 864 popular apps, finding that 40.6% contained a WebView associated with an ad library, and 10.8% (38) of the apps in their survey had an ad WebView capable of invoking Android code. The libraries they found responsible for this behavior confirmed our own results: all three were detected by AdRisk. However, as the survey focused on WebView use of any kind, they further discovered that WebViews allowed for several dangerous behaviors in core application code as well, as approximately 11% of developers appear to expose native app functionality to unauthenticated attackers on the web.

While the results of most of these surveys reconfirmed our initial findings, at least one relevant fact appears to have changed: the presence and prevalence of native code. Just last year, Afonso et al. [38] measured 1.2 million apps, finding that 446k potentially used native code – a far higher percentage than we observed in AdRisk, which may complicate matters of analysis and enforcement for future work. Indeed, the authors of FlexDroid [152] performed a smaller study of 100,000 apps, and found that 72% of the 295 ad libraries they discovered used dynamic code execution, and 17% now contained native code.
2.3.2 Sub-Application-Level Capability Leak Defenses

Fortunately, another class of systems have been proposed to address the problem of internal capability leaks within applications. Early works, such as AdSplit [156] and AFrame [189], move ad libraries out of the hosting app for isolation purposes, allowing the existing permission model to account for the different subject involved (i.e., distinct, lesser, permissions may be granted to the ad library alone). AdDroid [138] similarly moves advertising functionality out of the host app by introducing a new system service for this purpose.

The same intuition – move code of different provenance out of the app to achieve higher permission granularity – has been used in other contexts than simply ad libraries, highlighting the breadth of the problems posed by internal capability leaks. For example, LayerCake [147] extends the Android framework to allow different embedded principals to isolate their user interface components from one another; while ad libraries are one such principal mentioned in the paper, the work attempts to mitigate a different set of threats than those explored in AdRisk, such as clickjacking and ad fraud. Meanwhile, NativeGuard [164] isolates native third-party libraries, preventing a security problem in one from compromising the entire app. Our work exclusively focused on code written in Java; as native code is becoming more popular [38, 152], inevitably defenses must adapt to include it.

Instead of moving code out of the host app, a pair of works envision changing the Android framework to support finer-grained isolation, with the support of the developer. Compac [174] assigns each logical component within an app (e.g., an ad library) its own set of permissions, and tracks the provenance of each call to a permission-protected API. Unfortunately, it does not account for native code within each app, which could be used to circumvent its protections. FlexDroid [152] aims to solve the same problem, but uses hardware-based fault isolation to sandbox native third-party libraries, and additionally allows the app developer to specify a policy to deal with violations in any libraries contained in the host app. Case [195] removes the dependence on operating system support; while the developer must still cooperate, CASE uses hooking to retrofit module-level isolation atop the existing Dalvik VM.

Improving on the security guarantees of these previous three works, DroidDisintegrator [169] takes a two-stage approach to achieve fine-grained library isolation: first, the app in question is exercised while dynamic analysis based on TaintDroid [87] logs its flows; these flows are then embedded in the manifest, to generate component-level information flow policies for Android apps. At runtime, a modified Android framework isolates these components, grants them individual permissions, and enforces the policy about data flows between them. DroidDisintegrator has the security advantage that flows which are not exercised will not be permitted at runtime, due to its process-isolation model; however, this carries the concomitant risk of breaking app functionality if the analysis is incomplete.
2.3.3 Privacy-Preserving Ad Delivery

More generally, researchers have explored ways to deliver targeted ad content without disclosing any private information to the advertiser or ad network. For example, Adnostic [168] addresses the online ads and allows for behavioral web advertising without giving behavioral information to the ad network (by using a dedicated Firefox browser extension to prevent unnecessary information disclosure). πBox [119] does this at the level of an app, sandboxing it such that any potentially private information it requests can first be anonymized. MoRePriv [78] puts forward a system to allow privacy-preserving behavioral customizations for a variety of purposes, including ad delivery; LinkDroid [94], in a complementary move, extends the Android framework to try to measure the amount of privacy disclosure taking place and give users more options to control it.

MobiAd [103] instead approach by using a broadcast mode available to wireless providers to stream a large amount of tagged ad content that is then filtered by mobile devices. Privad [102] offloads ad selection to the client, but aims to do so in a way that is less disruptive to the existing industry model for ad networks; in particular, much emphasis (including a follow-on work [145]) is placed on preserving the auction mechanism by which advertisers compete for ad slots on the networks. These systems incorporate cryptographic billing to ensure that click-throughs are properly billed to the advertiser without compromising the consumer’s privacy, and without encouraging click fraud.

Lastly, there are some efforts that specifically aim to address the privacy concerns inherent with location information. Bindschaedler et al. [52] attempts to prevent tracking individual devices’ movements by changing their identifiers in crowded regions. PrivStats [142], MDD [143], PAMPAS [167], and Prio [72] all offer mechanisms so that aggregate location information can be irrefutably collected in a privacy-preserving way. While these systems are making progress in mitigating the privacy risks, it is unclear yet whether they can be applied in our context to handle the in-app ad security risks.
Chapter 3

Kernel-Level Privilege Separation

3.1 Background

Kernel rootkits are among the most insidious threats to computer security today. Embedding themselves within the operating system kernel, these rootkits enjoy unfettered access to the entire system and adopt various techniques to make themselves stealthy and “sticky,” thus preventing them from being detected and removed. Given the effectiveness of this approach, it is not surprising that there has been explosive growth in the number of new rootkit families in the years leading up to this work [33, 35]. Today, rootkits are often spoken about in the context of Advanced Persistent Threats, which couple them with a means of propagating themselves within a targeted network; these remain a major threat, being at the heart of major high-profile data breaches [27].

Due to the tempting position that the kernel occupies in most systems, it is often the center of a classical arms race. This work looks at a period of time when way in which the Linux kernel laid out its memory left it open to so-called “code-injection attacks,” where an attacker could inject their own into the kernel, then subsequently arrange for it to be executed. At that point, as the kernel does not have strong self-protection mechanisms, they would essentially have free reign over the entire system.

To avoid code-injection attacks, modern operating systems tend to hew to the $W \oplus X$ ("write XOR execute") security property, which states that a given memory page can be either writable or executable, but not both at the same time. Linux took some time to fully comply with that property; some of our other work [120] resulted in eliminating the last remaining writable-and-executable memory pages. Even so, it is still possible to build kernels today that do not properly enforce the $W \oplus X$ property.

Interestingly, the entire problem of code injection can be viewed as an unintended side-effect of the dominant computer architecture. Many of the first generation of computers represented
code and data differently, such that it was not possible for a data access to refer to code, or for a branch instruction to target data. This hard separation of code and data was characteristic of the Harvard architecture, in contrast to the dominant von Neumann architecture in use by most computer systems today, where code and data are both stored in the same memory system.

One possible solution to the problem of code injection, therefore, is to turn back the clock on computer architecture by emulating a Harvard architecture on a machine that is truly built according to the von Neumann model. NICKLE [146] took a software virtualization (i.e., binary translation) approach to emulate a Harvard architecture on x86, which essentially creates a separate memory space to reliably store authorized kernel code. By transparently redirecting kernel instruction fetches to the separate memory space, NICKLE is able to support unmodified kernels and guarantee their kernel code integrity, which effectively defeats most existing rootkits. However, the presence of a dedicated code memory and the need for transparent redirection of kernel instruction fetches require intercepting and redirecting every single kernel instruction execution, which unfortunately causes significant performance overhead.

Meanwhile, hardware virtualization extensions offer much greater efficiency, but existing works could only enforce $W \oplus X$ on cooperating kernels, whose memory was already laid out in accordance to the property. For example, SecVisor [153] proposes a tiny security hypervisor that solely enforces the $W \oplus X$ property on kernel memory pages. Given the profusion of legacy OS kernels that contain mixed kernel pages with both code and data [121, 122, 123], SecVisor modifies the kernel source code to make the OS kernel memory layout conform to the $W \oplus X$ property.

In this chapter, we introduce hvmHarvard, a hardware virtualization-based Harvard architecture on x86 that can not only transparently support commodity OSs without modification, but also effectively reduce the performance overhead. Specifically, we observe that the high performance overhead of implementing a software-based Harvard architecture is mainly caused by instruction-level interception and redirection (of kernel instruction fetches) to the code memory. As such, we propose page-level redirection in hvmHarvard so that the performance overhead can be significantly reduced without unnecessarily sacrificing the security guarantee.

There are two main challenges involved in changing from instruction-level redirection to page-level redirection of kernel instruction fetches. The first one comes from the fact that x86 is not designed to support the Harvard architecture. To address that, we make an unconventional use of split code and data TLBs on x86 in combination with recent hardware-based tagged TLB support [34]. In particular, with separate code and data TLBs, we can dynamically adjust the page table to virtualize the Harvard architecture on top of x86 (so that code and data each have its own memory address space). The tagged TLB support is essential here as it avoids flushing the code/data TLBs in a virtualized environment (e.g., VM exits – Section 3.3.1), thus allowing the hypervisor to safely intervene and manipulate the guest page table in use for
the Harvard architecture creation. With the separation of code memory and data memory, our Harvard architecture can naturally handle the mixed code and data pages in commodity OS kernels while still strictly enforcing $W \oplus X$. In the meantime, we also observe that the majority of existing kernel memory pages are not mixed. As a result, there is no need for hvmHarvard to keep a shadow copy of these pages, nor does it need to intervene on instruction fetches from them. By doing so, no processing overhead will be incurred on these pages and no extra memory space will be wasted, as they no longer need to be shadowed [146].

The second challenge stems from the need to perform mode-sensitive page-level redirection since we are interested in redirecting kernel instruction fetches only. In other words, we need to first determine the current running mode and then decide whether the corresponding instruction fetch should be redirected or not. This imposes a strict requirement to intercept every mode-switching event (e.g., including system calls) in the redirection logic. Intercepting these events at the hypervisor will cause significant performance overhead. Our solution to this problem involves altering the guest’s view of memory at each privilege level (or mode): all of user memory becomes non-executable when a process is executing at the kernel mode, and vice versa. For brevity, we call this a mode-sensitive view (Section 3.3.2). During the normal operation of the guest, hvmHarvard does not intercept and mediate the change between different views of memory. Instead, our system injects trampoline code to switch between these two views of memory upon the mode-switching event inside the guest. The trampoline mechanism leverages an Intel hardware virtualization extension called the CR3 Target Value List (Section 3.3.2) to avoid being trapped by the hypervisor and to achieve better performance.

We have implemented a Xen [49]-based proof-of-concept prototype. The prototype can transparently support a number of commodity systems including legacy Red Hat 8.0 (with a Linux 2.4.18 kernel) and recent Ubuntu 9.04 (running Linux 2.6.30-5). Our evaluation shows that our system is effective in preventing eight kernel attacks (including six real-world rootkits and two synthetic attacks) against legacy OS kernels that do not have the $W \oplus X$ support. Such protection is achieved with only a small performance overhead (i.e., < 5%). To summarize, this work makes the following contributions:

- We propose a hardware virtualization-based Harvard architecture to effectively protect commodity OS kernels from kernel rootkit attacks. Compared with existing approaches, our system can not only achieve a similar protection guarantee, but also significantly reduce the performance overhead suffered by previous approaches.

- The first key technique in our approach is page-level redirection of instruction fetches, which departs from prior efforts that perform instruction-level redirection. Our technique significantly reduces the performance overhead in the creation of the Harvard architecture on top of x86.
• The second key technique enables *mode-sensitive* redirection by redirecting *only* kernel instruction fetches. In this way, we can effectively avoid hypervisor intervention in the guest’s mode-switching events. As these events occur frequently inside the guest, this technique also contributes to reducing the overall performance overhead.

• Finally, we present a Xen-based system prototype. The evaluation results with the prototype confirmed the practicality and effectiveness of our approach.

The rest of this chapter is structured as follows. First, we briefly describe necessary technical background on the Harvard architecture and hardware virtualization in Section 3.2. Our system design and implementation are then presented in Section 3.3 and Section 3.4, respectively. After that, we present the evaluation results in Section 3.5, which is followed by the discussion on possible limitations and their improvement in Section 3.6. Finally, we summarize the work in Section 3.7.

### 3.2 Key Concepts

In this section, we briefly review some key concepts that are essential to our system but may be unfamiliar to some readers: the Harvard architecture and shadow paging in virtualization. Readers with sufficient background can safely skip this section.

#### 3.2.1 Harvard Architecture

![Figure 3.1: The Harvard architecture](image)

Modern computers use a single address space to refer to working memory. This model of memory is commonly known as the von Neumann architecture. Interestingly, some of the very earliest computers used two utterly separate working memories, one for instructions and one for data. This arrangement is known as the Harvard architecture (Figure 3.1). In a pure
Harvard architecture machine, data accesses and instruction accesses are treated as accessing totally distinct address spaces. From a security standpoint, this addressing scheme eliminates code injection attacks. For example, some buffer overflow attacks use an overlong memory copy operation to overwrite memory that will be executed as code. A pure Harvard architecture machine is not vulnerable to this class of attacks, as their addressing scheme does not allow code and data to be referred to interchangeably.

This work focuses on a widely deployed processor family, x86, which has a unified address space for main memory and is thus a von Neumann architecture. However, x86 processors typically have separate caches for instructions and data. When executing from cache, the processor behaves like a Harvard architecture machine\(^1\). Only when main memory must be consulted does x86 look like a von Neumann architecture. This observation is the foundation of our page-level redirection technique for the creation of the Harvard architecture on top of x86 (Section 3.3.1).

3.2.2 Virtualization and Shadow Paging

Virtualization involves running a guest operating system in an environment that provides the illusion of complete access to a physical machine. All the resources used to construct such an illusory machine constitute a Virtual Machine (VM), while the software that maintains one or more VMs is known variously as a hypervisor or a Virtual Machine Monitor (VMM). The hypervisor is commonly considered to be part of Trusted Computing Base (TCB) as it is strictly isolated from the VMs it manages and is often much smaller than modern operating systems.

There are several ways to virtualize a guest operating system. Since our work is based on hardware virtualization, we focus on its operation here. In particular, based on certain processor extensions, hardware virtualization operates a “trap-and-emulate” model. When a guest OS wishes to perform a privileged operation, the hardware has two options: either it can handle the request based on the processor extension for hardware virtualization, or if that is not possible, it can pass control to the hypervisor for handling. Handling the latter case constitutes a goodly portion of the hypervisor’s workload and is typically an involved process.

Shadow paging is one such example. To better describe it, we first review how memory management works on an un-virtualized machine. Recall that x86 supports two memory protection mechanisms: segmentation and paging. They protect memory in a similar way by essentially permitting a higher-privilege piece of software to put blinders on a lower-privilege program, thus restricting its view of memory to only those things it is supposed to be able to access. Since segmentation support is being phased out in the new 64-bit long mode, we focus on the paging protection mechanism. In essence, paging uses page translation tables, or page tables for short, to remap memory for a given process. Virtual addresses are translated into physi-

\(^1\)This hybrid architecture is known as a modified Harvard architecture; many processors with the caching feature use such an arrangement today.
Virtual addresses by these tables. These tables are also used by the hardware to enforce certain permissions policies (e.g., NX [32]) on the types of accesses allowed.

Virtualization has not changed this picture of the process; it has merely added another layer underneath it. By leveraging paging, the hypervisor divides the machine’s memory into distinct logical machine memories. The guest OS in a VM then treats the memory it is given in the traditional way, dividing it up between the applications running in the guest. Under hardware virtualization, however, the OS itself does not know the real machine addresses that make up its allotted memory. With shadow paging, the hypervisor solves this problem by introducing an extra layer of indirection. In particular, a shadow table is created for a guest and maintained in the hypervisor. An unsuspecting guest OS kernel is allowed to maintain its own page tables, but they are not actually used by the hardware. Instead, the hypervisor marks these guest page tables read-only. Any attempt to write to them therefore generates a page fault, which is trapped by the hypervisor. The hypervisor, in turn, emulates the write request, eventually outputting the equivalent entry into the “real” page table used by the hardware. The guest can never see this real page table, which is assiduously kept synchronized with the one it can see – thus the name “shadow page table.”

This arrangement is illustrated graphically in Figure 3.2. In the diagram, a virtual address (VA) is translated through both the guest’s and the hardware’s page tables. The guest’s page tables eventually lead to a guest physical address (GPA) – the address the guest thinks of as being a hardware address. The shadow page tables instead translate the same virtual address...
into the real machine address (MA). The tables are kept synchronized by the hypervisor; this synchronization is represented by the dotted lines in the figure.

3.3 Design

In this work, we aim to develop a hardware virtualization-based Harvard architecture that can efficiently support unmodified legacy OS kernels and protect them from kernel rootkit attacks. Specifically, the presence of two distinct memory spaces for code and data in a Harvard architecture is useful for blocking code injection attacks and enforcing the $W \oplus X$ property. In this work, we propose to take a step further by enforcing mode-sensitive $W \oplus X$, also known as $W \oplus KX$. Due to our focus on OS kernel protection, $W \oplus KX$ requires that a user-level memory page will not be executable from the kernel mode and vice versa. Commodity hardware by default allows the execution of user-level memory pages at kernel privilege, which opens up “interesting” opportunities for kernel rootkit infection. As our defense, $W \oplus KX$ is proposed to not only enforce $W \oplus X$, but also effectively block this infection vector.

**Threat Model and System Assumption**  In this work, we assume an adversary model where attackers or kernel rootkits are able to exploit software vulnerabilities in an OS kernel to launch code injection attacks. Accordingly, we also assume kernel rootkits have the highest privilege level inside the victim VM (e.g., the root privilege in a UNIX system) and have full access to the VM’s memory space (e.g., through `/dev/mem` in Linux). However, the goal of a kernel rootkit is to stealthily maintain and hide its presence in the victim system; to do so, it will need to execute its own (malicious) code in the kernel space. We note that such a need exists in most kernel rootkits today, and we will discuss possible exceptions in Section 3.6.

In the meantime, our system assumes a trustworthy hypervisor as the necessary trusted computing base (TCB) to provide strict VM isolation. This assumption is shared by many other hypervisor-based security research efforts [84, 96, 111, 126, 177] and being hardened by existing hypervisor-protection solutions [131, 175]. We will discuss possible attacks (e.g., VM escape) in Section 3.6. With this assumption, we consider the threat from layer-below attacks launched from physical hosts outside of the scope of this work.\(^2\)

3.3.1 Page-Level Redirection for $W \oplus X$

The central scheme of our approach is to efficiently create a Harvard architecture (Figure 3.3) on x86 by virtualizing one memory space for code and another for data. To achieve our goal, we observe the presence of separate TLBs for instruction fetches and data accesses. Note that each

\(^2\)There exists another type of layer-below or specifically hardware DMA attack that is initiated from within a guest VM. However, since the hypervisor itself virtualizes or mediates guest DMA operations, recent hardware support for IOMMU can be readily adopted to intercede and block them. Therefore, we do not consider them in this work.
Figure 3.3: Page-level mode-sensitive redirection enables an efficient implementation of the Harvard architecture on top of x86.

TLB entry caches the translation result from a virtual address to a physical address. When a memory access or an instruction fetch occurs, the virtual address lookup will go through the corresponding TLB first. Should that TLB not contain an entry for the requested translation (called a TLB miss), the hardware walks through the page table entries in main memory to do the lookup, then constructs such an entry. As a result, from the TLB’s perspective, the hardware itself thinks in terms of two address spaces. However, in normal operation, these address spaces are kept synchronized and thus describe a unified memory space. Fortunately, to our benefit, there is no hardware requirement that this must be the case. In other words, to emulate a pure Harvard architecture, we can take advantage of these two TLBs by desynchronizing and loading them with two different page table entries for the same virtual address, thus creating two distinct memory spaces for code and data.

Unfortunately, the de-synchronization of these two TLBs is a delicate process, which is complicated by the fact that a TLB entry has a relatively limited lifespan. First, the TLBs are not large enough to cache all translation results at the same time, which means that older entries are eventually overwritten by newly-requested translations. Second, when an OS kernel either alters a page table or switches address contexts, these caches are implicitly flushed. Third, x86 provides very few instructions for interacting with the TLBs. In fact, after enabling the paging mode, the provided instructions are mainly used for removing one or all entries from both TLBs, which means the only way for us to populate a TLB entry will be by performing an address translation that eventually winds up in that cache.
**Input:** Redirected Page Address (addr), Pagetable Entry for addr (pte)

```c
/* handling NX-based page fault */
pte = the_code_page (addr);
set_trap_flag ();
return_to_guest ();

/* handling TF-based fault */
pte = the_data_page (addr);
unset_trap_flag ();
return_to_guest ();
```

**Algorithm 1:** TLB de-synchronization algorithm.

To deal with the above challenges, we need to effectively intercept the hardware’s attempts to re-populate TLBs. In particular, for the virtual addresses of interest, when there is a TLB miss, the hardware consults the page table and checks the permission bits of the entry it loads. If those permissions are violated, a page fault (or **#PF**) exception will be thrown. When there is a TLB hit, the cached entry’s permissions are directly checked without consulting the page table. As a result, in the case of a TLB miss, we need to carefully prepare the page table in a way that will load the desired translation results as well as related permissions into respective TLBs.

There are three permission bits that can cause useful faults: the **USER** bit, the **PRESENT** bit and the **NX** bit. The **USER** bit only faults when a user-mode instruction fetch references a kernel page. With our focus on kernel protection, we are not interested in using this bit. The **PRESENT** bit, if not set, traps any access – which would lead to many expensive world switches. The **NX** bit causes a fault on any instruction fetch from pages with this bit set. In our system, we naturally leverage the **NX** bit.

In particular, to use the **NX** bit to cause one virtual address to map to two context-sensitive memory pages, we map the address to its data memory page and set its **NX** bit. If execution branches to an address within the page, the page fault handler substitutes its entry to code memory page and clears the **NX** bit. In order to load the entry into the instruction TLB (ITLB), the page fault handler must allow the guest to execute an instruction using this entry. However, once the code page entry has been loaded, the system needs to regain control to restore the map back to the data memory page. If this is not done, the data TLB (DTLB) may wind up being populated with the code page entry, routing data reads to the code page and thus violating the Harvard architecture. Note that the code page entry is marked as **read-only** and there is no way to cause a page to be executable yet not readable on the x86 architecture.

To ensure that the page table is restored to the corresponding data entry as soon as possible, our design relies on the x86 single-step execution feature. Specifically, by setting the **trap** flag (or **TF**) of the **EFLAGS** register, the processor will generate an exception after every instruction. This feature allows us to execute one instruction, and then restore the data page entry in the **TF** handler. The process is shown in pseudo-code in Algorithm 1.
In this way, our design can populate the ITLB with one record and DTLB with another record without interfering each other. Here, we point out that if by trapping the execution of a guest VM to the hypervisor, a VM exit (or \texttt{VMEXIT}) occurs. In some processors, VM exits will flush the TLBs, which defeat our purpose of de-synchronizing TLBs. In our prototype, we leverage a hardware feature called tagged TLB \cite{34} that is available in all recent hardware-virtualized AMD processors as well as Intel processors based on the new Nehalem architecture. This hardware feature essentially adds an extra field or an identification “tag” to each TLB entry that specifies the VM context within which the entry is valid. When a VM exit occurs, these entries will not be flushed. More details about our system will be presented in Section 3.4.1.

3.3.2 Mode-Sensitivity for $W \oplus KX$

By effectively creating a Harvard architecture on x86, our page-level redirection technique is able to enforce $W \oplus X$ while accommodating mixed kernel pages in commodity OS kernels. However, the $W \oplus X$ enforcement is still insufficient due to the need to block the execution of user-level pages from the kernel level. In other words, we need to enforce a stronger $W \oplus KX$ policy. As mentioned earlier, this is necessary as commodity OS kernels disallow the access of kernel memory pages from user mode, but do permit the execution of user memory pages from kernel mode.

To elaborate on this, the x86 architecture has two related concepts in this vein: the \texttt{USER} page table permission bit and the Current Privilege Level (CPL) bits in the \texttt{CS} register. The CPL simply determines what instructions are valid – including access right checking on instruction fetches. The most-privileged CPL (or ring 0 where the kernel runs) has all the capabilities of the least-privileged CPL (or ring 3 where user-level applications run). Therefore, while it is illegal for a program executing at the ring 3 privilege to access kernel space, it is perfectly acceptable for a ring-0 kernel to branch its execution to user space.

With $W \oplus KX$, we aim to define a new Kernel eXecute (KX) mode of operation. In this mode, instruction fetches only succeed if the privilege level of the machine matches the privilege level of the page table entry. In other words, if \texttt{USER} is cleared for a page table entry, it is only executable at CPL=0, and when \texttt{USER} is set, it is only executable at CPL=3.

To achieve this, we propose maintaining \textit{two} shadow page tables instead of one in the normal situation: one for user-privilege (or mode) execution and one for kernel-privilege execution. Each has the \texttt{NX} bit set for the opposite privilege’s pages. A straightforward approach would require the hypervisor to intervene and swap the shadow page table upon every mode switch, from user to kernel and vice versa. Unfortunately, this scheme would induce a large number of costly \texttt{VMEXIT}s – two for every system call. To reduce this overhead, note that modern processors introduce special instructions – \texttt{sysenter/sysexit} to enable fast transfers between
user and kernel. As these instructions use registers to point to the entry point of the system call handler, by redirecting that register to our trampoline code, we can handle a large number of mode switches in a performance-efficient fashion. More specifically, our approach leverages a hardware feature known as the “CR3 Target Value List.”[70] This feature is designed to allow a hypervisor to whitelist a set of expected CR3 values: when a guest changes CR3 to one of these values, the hypervisor is not consulted, saving a significant number of cycles that would be wasted on a world switch. In our prototype, our system injects a trampoline into the guest that simply switches page tables upon each mode switch, before the actual OS system call handler is invoked. Similarly, we use this trampoline to switch the page tables again before the system call handler returns back to user mode.

We assert that this optimization does not harm the $W \oplus KX$ security guarantee offered by our system. Specifically, the trampoline code is located on a page that the hypervisor prevents the guest from modifying. Also, if the guest invokes the trampoline code in an unintended way, it will always wind up either transferring control to the sysenter/syscall handler or executing the corresponding return instruction. From the OS kernel’s perspective, the $W \oplus X$ property is not violated. More detailed discussion will be presented in Section 3.6.

Finally, it is worth mentioning that our system follows the same steps proposed in NICKLE to support loadable kernel modules (LKMs) [146]. In particular, we simply verify the hash signature of such drivers (and the main kernel) when they are being loaded. For example, for Linux kernels, we leverage the fact that the kernel’s module loader calls the init() method of a module when it is being loaded. As this will cause a page fault due to our page-redirection technique, we can check the instruction pointer (IP register) to see if it matches an address within the kernel’s module loader. If it does, the system can locate the module definition structure and use that information to determine how to verify the module. Falsifying the module structure information would inevitably result in a hash signature inconsistent with the trusted version of the module, causing the falsified module to be simply rejected by our system. Note that we do not need to modify the guest operating system; our system simply needs to know how to find the information it needs in the guest operating system’s memory. Such knowledge can be provided in a number of ways, e.g., either directly compiled into the hypervisor, loaded in the VM’s metadata or indirectly hinted to the hypervisor from a hypercall within the VM.

### 3.4 Implementation

We have developed a proof-of-concept prototype on top of Xen 3.3.1, targeting fully-virtualized 32-bit legacy guests running under a 32-bit PAE hypervisor. Our development was tested against a Red Hat 8.0 image (running a Linux 2.4.18 kernel) and an Ubuntu 9.04 image (running a Linux 2.6.30-5 kernel). Our development machine had a Core i7-930 Nehalem processor with
recent hardware virtualization support. Our current prototype only supports a single virtual CPU for one guest and the support of SMPs are left to future work. In the following, we present additional implementation details for the two key techniques in our approach.

3.4.1 Page-Level Redirection

As mentioned earlier, our scheme virtualizes a pure Harvard architecture machine on x86 by using a hypervisor to desynchronize the processor’s TLBs. Naturally, our prototype mainly deals with various particulars of the x86 paging mechanism and related TLB operations. In particular, our experience indicates that there is a strong correlation between the frequency with which the TLBs must be fixed up and the performance overhead of the system as a whole. Note the process of de-synchronizing or splitting a page’s TLB entries is a costly operation. Each time a page needs to be split, there are two associated VMEXITs: one caused by the NX-based page fault to populate the ITLB, and another from the single step fault handler to populate the DTLB. Because of that, it is critical to avoid generating these events if possible.

In our prototype, we implement an optimization that is akin to the traditional copy-on-write (COW) technique. Recall that one main purpose of our system is to ensure \( W \oplus X \). As such, if some kernel pages in commodity OSs are already amenable for \( W \oplus X \) enforcement, we can simply enforce it without needing to create two separate copies (one for code and one for data) in the first place. By doing so, we can not only avoid allocating additional memory spaces in storing copies, but also reduce the number of VMEXITs that would otherwise be needed to maintain the separate presence of code and data copies.

To further elaborate that, consider the impact of splitting a kernel page \(^3\). If the kernel page is never used as code, the additional overhead will be incurred when generating and maintaining the two copies, though there is little or no performance impact. However, if the kernel page is never used as data, then we will be splitting the page every time it is executed and the translation is not cached in the ITLB (or already flushed from the ITLB). As mentioned earlier, this process will involve the hypervisor and cause VMEXITs, resulting in a high performance overhead.

In our prototype, to determine the liveness of a kernel page, we perform basic reference-counting and dynamically track the number of times a given kernel page is referenced by the guest’s page tables. In addition, by counting the number of writable mappings to a given kernel page, our system can intelligently choose not to split the page if that count is zero. In this way, we can further avoid unnecessary VMEXITs for better performance.

\(^3\)Xen’s concept of kernel pages can be different than the guest OS’. For example, Xen does not internally use 2M or 4M “superpages”; if the guest OS allocates these, Xen treats them as a large number of normal 4K pages.
### 3.4.2 Mode-Sensitivity Support

To make the page-level redirection mode-sensitive, we implement two shadow page tables: one for guest user mode and another for guest kernel mode. As a result, every time the guest OS wishes to make a change to its page tables, the hypervisor intercepts the change and synchronizes it with the two shadow pages. As synchronization will require the hypervisor to walk through the shadow page tables and make the corresponding hardware-visible change, the presence of two shadow page tables will double the cost of synchronization. To reduce the cost, our prototype opts to interleave two page tables; this allows a single walk through them to find both entries related to a particular page table update. Specifically, for each page table bifurcated in this way, twice the normal amount of memory for shadow page tables is allocated. The low-order version of the page table is used for the guest kernel mode, and the high-order version is for the guest user mode. With that, one walk is needed to find the location to alter, followed by a privilege-level check that determines which changes to make and where to look for the second copy of that page.

With the two shadow page tables in place, our prototype further takes another optimization. Considering the fact that page tables are laid out in a layered hierarchy, we can trade granularity for ease of updating, simply by having two distinct top-level page tables map down to the same set of level-1 page tables (see Figure 3.4). The top levels of the page table are not altered as frequently as the lower levels are, leading to disproportionately less update overhead. They are also smaller (as there are fewer such top-level entries), leading to less cache pressure when compared to the case where all entries had to be maintained separately. Using a 32-bit Linux guest as an example, the Linux kernel occupies the top one gigabyte of address space. As the shadow page tables are 32-bit PAE tables, this neatly corresponds to one of the four top-level entries. Though the top-level entries do not have the `NX` permission bit, we can maintain two sets of the level-2 page tables instead that have the `NX` permission bit.

Afterwards, the two shadow pages will be switched based on the current running mode of the guest VM. In our prototype, we hook the handler for the `sysenter` instruction (by detouring the corresponding Model Specific Register or MSR content) to capture the user-to-kernel mode switch. Similarly, we also detour the `sysexit` execution by performing a kernel-to-user switch. We point out that such detouring happens inside the guest context with a trampoline without involving the hypervisor, thus avoiding unnecessary VMEXITs. However, from another perspective, our prototype can still function properly without hijacking them because the hypervisor will simply step in and switch page tables itself, though at a lower pace.

An astute reader may observe that the trampoline code will essentially change CR3, the page table base address register. Changes to CR3 will typically be trapped by the hypervisor. Fortunately, a recent hardware feature, i.e., the CR3 Target Value List, allows our page table
switch without being trapped by the hypervisor if the new CR3 value is on the target value list. However, the CR3 update is still considered a context switch, which unfortunately causes an unnecessary TLB flush – purging any split entries from the instruction TLB. Interestingly, the related level-1 page table entries contain a GLOBAL bit that can prevent a TLB flush from purging a particular entry.

There is a subtle issue in the interplay between the CR3 Target Value List and the GLOBAL bit. By definition, the hypervisor is not alerted if CR3 is changed to a value on the list. Likewise, if a split entry in the TLB is not purged, the page tables will not be consulted upon an instruction fetch to its virtual address. Therefore, if our user-mode CR3 value is loaded from a page that is marked GLOBAL, execution could branch to user land while still at high privilege! Fortunately, there are only two ways that CR3 can take a new value: via hardware task switching (ltr) or through the explicit assignment (mov cr3, <general register>). Hardware task switching is not used by either Windows or Linux. For the more common mov cr3 operation, we ensure that the instruction pointer, after a mov cr3, <register> operation, will always point to a virtual address that does not map to a TLB entry with the GLOBAL bit set. To assure that, we can scan each page as it is being split, ensuring that the opcode for this dangerous operation does not occur. In other words, we look for that string of bytes throughout the split page. If it is found, the split code will ensure that upon every insertion to the ITLB, that split page’s entry will not have the GLOBAL permission bit set.

---

4Note that even if it is used, the ltr operation acts on tables that are privileged and hardware virtualization allows for trapping the ltr operation. In other words, we can still prevent hardware task switching from breaking our W @ KX guarantee.

5Note that there are a few corner cases worth mentioning. The mov cr3, <register> instruction is translated to 0f 22 d? in machine code. If the split page ends neatly with 0f 22 d?, then it would put the instruction pointer onto the next page, whose GLOBAL property is uncertain. Fortunately, that case does not occur in the

---

Figure 3.4: Two shadow page tables: the user-mode page table and the kernel-mode page table share the same level-1 entries, but not top-level and level-2 entries.
Table 3.1: Effectiveness of our system

<table>
<thead>
<tr>
<th>Rootkit</th>
<th>Attack Vector</th>
<th>Prevented?</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>adore-ng 0.56</td>
<td>LKM</td>
<td>Yes</td>
<td>Module fails to load</td>
</tr>
<tr>
<td>superkit</td>
<td>/dev/kmem</td>
<td>Yes</td>
<td>Crashes</td>
</tr>
<tr>
<td>mood-nt 2.3</td>
<td>/dev/kmem</td>
<td>Yes</td>
<td>Crashes</td>
</tr>
<tr>
<td>sk2rc2</td>
<td>/dev/kmem</td>
<td>Yes</td>
<td>Crashes</td>
</tr>
<tr>
<td>eNYeLKM 1.2</td>
<td>LKM</td>
<td>Yes</td>
<td>Module fails to load</td>
</tr>
<tr>
<td>Phalanx b6</td>
<td>/dev/mem</td>
<td>Yes</td>
<td>Crashes</td>
</tr>
<tr>
<td>synthetic-1</td>
<td>LKM</td>
<td>Yes</td>
<td>Module fails to modify itself</td>
</tr>
<tr>
<td>synthetic-2</td>
<td>LKM</td>
<td>Yes</td>
<td>insmod crashes</td>
</tr>
</tbody>
</table>

3.5 Evaluation

To test the effectiveness of our prototype, we run six real-world rootkits and two synthetic exploits (both violate $W \oplus KX$) against a default Ubuntu 9.0.4 system. These attacks were selected as representative of the infection vectors used by existing kernel rootkits. In every case, our system was able to defeat the infection and protect the system. In the following, we present details of two representative experiments.

**Mood-NT Rootkit Experiment** Some rootkits install themselves by directly writing to mixed pages in kernel memory. In this experiment, the mood-nt rootkit [146] uses the /dev/kmem interface to access kernel memory through the file system. Specifically, the rootkit uses the interface to copy its resident logic into kernel memory, and then overwrites function pointers to hijack the kernel’s control flow.

When the test system is protected under our prototype, code injection appears to work fine as the injected content is directly written into the data page. However, when one of the rootkit’s function pointers is called, our page-level redirection technique immediately causes the resulting instruction fetch to a code page, not the data page that contained the injected content. As a result, instead of fetching the rootkit’s code, the processor attempts to execute whatever is in the code page, eventually leading to a crash in our experiment.

**Synthetic Attacks** In this experiment, we intentionally play with the $W \oplus KX$ protection by redirecting kernel control flow to user-space code. Since we do not have a rootkit sample that was developed in this way, we simply synthesize an attack that would execute user code at kernel privilege.

Specifically, we implemented a branch-to-userspace exploit as a loadable kernel module. In the module’s initialization function, we create a pointer to an address within insmod’s address space. This address in user space contains an instruction sequence that copies the top of the Linux kernels we have examined. Such a special case can also be handled upon insertion into the TLB, by proactively re-populating the next page’s TLB entry as $\neg$GLOBAL.
Table 3.2: Software configuration for performance evaluation

<table>
<thead>
<tr>
<th>Item</th>
<th>Version</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ubuntu</td>
<td>9.0.4</td>
<td>Using Linux 2.6.30</td>
</tr>
<tr>
<td>Apache</td>
<td>2.0.59</td>
<td>Using the default high-performance configuration file</td>
</tr>
<tr>
<td>Kernel</td>
<td>2.6.30</td>
<td>Standard kernel compilation</td>
</tr>
<tr>
<td>ApacheBench</td>
<td>2.0.40-dev</td>
<td><code>ab -c3 -t 60 &lt;url/file&gt;</code></td>
</tr>
<tr>
<td>LMbench</td>
<td>3.0alpha</td>
<td>Using the default configuration</td>
</tr>
</tbody>
</table>

stack into EBX and then returns. Therefore, after successfully executing it, EBX should equate to EIP. Running under hvmHarvard, the execution faults to the hypervisor when the first user instruction is fetched. From the page fault handler, it reports the fault as a NX violation and relays it to the guest OS kernel, which then terminates the insmod process.

**Performance Overhead** To evaluate the impact on system performance, we have performed benchmark-based measurements. In particular, we use two application-level benchmarks and one microbenchmark to evaluate the system. They are (1) a normal compilation of the Linux 2.6.30 kernel, (2) network throughput test on the Apache web server using the ApacheBench [40], and (3) a standard system benchmark toolkit called LMbench [125]. Our tests were performed on a Dell Optiplex, which runs the Ubuntu 8.04 system and has an Intel Core i7-920 (2.66GHz) CPU and 4GB RAM. The guest VM runs Ubuntu 9.04 with Linux kernel 2.6.30-5 and 1GB of memory. For comparison, we run the guest VM on Xen 3.3.1 twice, with and without protection. The software configuration for our evaluation is shown in Table 3.2. The benchmark programs were run ten times and averaged. Our results are shown in Table 3.3.

Table 3.3: Application benchmark results. For make, lower is better; for Apache, higher is better.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Without protection</th>
<th>with protection</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>make kernel</td>
<td>41.289 s</td>
<td>43.312 s</td>
<td>4.9%</td>
</tr>
<tr>
<td>ApacheBench</td>
<td>11728.68 req/s</td>
<td>11497.24 req/s</td>
<td>2.0%</td>
</tr>
</tbody>
</table>

In our first application benchmark, we compiled our guest VM’s kernel with the command ‘make kernel’, using time to measure how long the process took. The system under protection takes 44.275 seconds to complete, which is 4.9% longer than the compilation time in an unprotected system. In our next application benchmark, we set up an Apache [39] web server. The ApacheBench program, ab, was run against a small (15K) html file on that server. We then collected the network throughput and the results show a 2.0% slowdown. We also evaluated our system with LMbench [125], which is a micro-benchmark for OS kernel performance. The tasks include process creation, basic arithmetic operations, context switching, file system operation, local communication, and memory latency. Among these results, the maximum overhead of our
system is 4.70% when doing context switching. The overhead comes from updating the CR3 Target Value List that is used for later switching of the two shadow page tables. Other tasks such as performing basic arithmetic or floating-point operations incur the lowest overhead, which is nearly zero.

### 3.6 Discussion

In this section, we discuss several issues related to our system. First, our goal here is to efficiently create a Harvard architecture on x86 and enable $W \oplus KX$ for kernel code integrity protection. As a result, our system is not able to protect the kernel control-flow integrity. In other words, an attacker could possibly launch a “return-into-libc” style attack or the so-called return-oriented attack [54, 108, 154] within the kernel by leveraging only the existing authenticated kernel code. Fortunately, solutions exist for protecting control flows [37, 101, 140, 176] and data flow integrity [61] for user-level applications, which could be potentially extended to complement our system for kernel protection.

Second, as with existing systems for kernel code integrity, our current implementation does not support self-modifying kernel code. This limitation can be removed by intercepting the self-modifying behavior (e.g., by trapping and validating the self-modification behavior) and re-authenticating and updating the kernel code in the code memory after the modification.

Third, our system currently does not support kernel page swapping. Linux does not swap out kernel pages, but Windows does have this capability when under heavy memory pressure. Supporting kernel page swapping would require intercepting swap-out and swap-in events and ensuring that the page being swapped in has not been maliciously tampered with.

Fourth, hvmHarvard cannot take advantage of the hardware-assisted paging mechanisms built into modern AMD and Intel processors [34, 70]. These schemes do not require the hypervisor to intervene when the guest wishes to alter its page table (as in shadow paging), resulting in superior performance. Unfortunately, our page-level redirection scheme requires page table updates be registered with the hypervisor. Consequently, further work would be required to adapt our scheme to use hardware-assisted paging.

Finally, we point out that our scheme assumes a trustworthy hypervisor to enforce $W \oplus KX$. This assumption is needed because it essentially establishes the root-of-trust of the entire system and secures the lowest-level system access. We also acknowledge that a VM environment can potentially be fingerprinted and exploited [115, 149] by attackers. Fortunately, recent solutions on hypervisor protection [116, 124, 175] can be employed to thwart these attacks. Also notice that as virtualization continues to gain popularity, the concern over VM detection may become less significant as attackers’ incentive and motivation to target VMs increase.
3.7 Summary

In this chapter, we present hvmHarvard, a hardware virtualization-based, efficient implementation of the Harvard architecture on top of x86. The x86 architecture is a von Neumann architecture by design, which efficiently shares memory storage between code and data. Unfortunately, the way in which this architecture was used by many operating system kernels led to code injection attacks, as these kernels did not properly use the memory protection features of the x86 architecture. In contrast to the problematic von Neumann architecture, the Harvard architecture has two memory spaces (one for code and one for data) and is thus inherently robust to code injection attacks employed by many contemporary kernel rootkits. Since the architecture of many operating system kernels did not distinguish between the privileges necessary to change data and the privileges to change code, even though it is very unusual to treat code as data or vice versa; we therefore can safely virtualize a Harvard architecture atop a von Neumann one to block these attacks. Prior efforts used instruction-level redirection to virtualize the Harvard architecture, while our approach proposes a page-level, mode-sensitive scheme to achieve the same goal but with a significantly reduced performance overhead. We have implemented a Xen-based prototype. Our evaluation shows that it allows for transparent support of legacy OSs (without modification) as the guest and protects them from existing kernel rootkit attacks with a small performance overhead (< 5%).
Chapter 4

Application-Level Capability Leaks

4.1 Background

As discussed in Chapter 1, Android uses a permission-based security model to inform the user about the potential capabilities of each app as it is installed. In other words, they allow a user to weigh the app’s capability against its stated purpose to determine whether or not to install it. If apps were able to access these permissions without having to explicitly request them, the central conceit of the security model would be undermined.

Unfortunately, there are apps on every Android device that the user did not install themselves: the preloaded apps that make up the phone’s firmware. Given that many phones are “locked” by the carrier, and that the window of support for any given phone is often quite short, any external capability leak present in phone firmware is likely to persist for a very long time. Adoption of new releases can be slow: at the time of this writing, the latest major Android release is version 7.0 (“Nougat”), which was released in August of 2016, with a minor feature update (version 7.1) following in October. As of June 20th, 2017, only approximately 8.9% of active Android devices are running version 7.0, with a further 0.6% running version 7.1 [4], despite it touting having a number of new security features and improvements [25].¹

In this chapter, we systematically study eight popular Android smartphones from leading manufacturers, including HTC, Motorola, and Samsung and are surprised to find out these stock phone images do not properly enforce the permission-based security model. Specifically, several privileged (or dangerous) permissions that protect access to sensitive user data or phone features are unsafely exposed to other apps which do not normally have access to them, due to not having the necessary permission themselves.

¹ Fortunately, since this work was performed, Google and several major smartphone vendors have committed to monthly security updates [1, 20, 24]. However, problems such as the ones reported here are manufacturer-specific, and thus the speed with which such issues are addressed remains sadly variable.
To facilitate exposing external capability leaks, we have developed a system called Woodpecker. By employing data flow analysis on pre-loaded apps, Woodpecker systematically analyzes each app on the phone to explore the reachability of a dangerous permission from a public, unguarded interface. To better examine possible capability leaks, our system distinguishes two different categories. *Explicit capability leaks* allow an app to successfully access certain permissions by exploiting some publicly-accessible interfaces or services without actually requesting these permissions by itself. *Implicit capability leaks* allow the same, but instead of exploiting some public interfaces or services, permit an app to acquire or “inherit” permissions from another app with the same signing key (presumably by the same author). Consequently, explicit leaks represent serious security errors as they subvert the permission-based security model in Android while implicit leaks could misrepresent the capabilities available to an app.

We have implemented a Woodpecker prototype to uncover both types of capability leaks in Android-based smartphones. Our current prototype focuses on 13 representative privileged permissions that protect sensitive user data (e.g., geo-location) or phone features (e.g., the ability to send SMS messages). We have used our prototype to examine eight popular Android phones: HTC Legend/EVO 4G/Wildfire S, Motorola Droid/Droid X, Samsung Epic 4G, and Google Nexus One/Nexus S. Our results show that among these 13 privileged permissions, 11 were explicitly leaked, with individual phones leaking up to eight permissions.\(^2\) In particular, by exploiting these leaked capabilities, an untrusted app on these affected phones can manage to wipe out the user data on the phones, send out SMS messages (e.g., to premium numbers), record user conversation, or obtain user geo-locations – all without asking for any permission.

The rest of this chapter is organized as follows: Section 4.2 and Section 4.3 describe our system design and implementation, respectively. Section 4.4 presents the detailed evaluation results from our study of eight Android smartphones. Section 4.5 discusses the limitations of our approach and suggests possible improvement. Finally, Section 4.6 summarizes our findings.

### 4.2 System Design

We aim to identify capability leaks, i.e., situations where an app can gain access to a permission without actually requesting it. Each such situation essentially sidesteps Android’s permission-based security model. In this work, we choose to focus on those permissions used by the pre-loaded apps as a part of an Android phone’s firmware, since the firmware has access to some permissions that are too privileged to be granted to third-party apps. For simplicity, we use the terms “permissions” and “capabilities” interchangeably.

\(^2\)Starting in April, 2011, we began reporting the discovered capability leaks to the corresponding vendors. Motorola and Google were very quick to confirm all discovered vulnerabilities related to their phones. Unfortu-
Figure 4.1: Detecting external capability leaks with Woodpecker

Figure 4.1 provides a high-level overview of our system. To detect the two different kinds of capability leaks (i.e., explicit and implicit), our system performs two complementary sets of analysis. Specifically, to expose explicit leaks of a capability, our system first locates those (pre-loaded) apps in the phone that have the capability. For each such app, our system then identifies whether a public interface is exposed that can be used to gain access to it. (This public interface is essentially an entry point defined in the app’s manifest file, i.e., an activity, service, receiver, or content provider.) In other words, starting from some public interface, there exists an execution path that can reach some use of the capability. If this public interface is not guarded by a permission requirement, and the execution path does not have sanity checking in place to prevent it from being invoked by another unrelated app, we consider the capability leaked. Our system then reports such leaks and further provides evidence that can be used to fashion input to exercise the leaked capability.

On the other hand, implicit capability leaks arise from the abuse of an optional attribute in the manifest file, i.e., “sharedUserId.” This attribute, if defined, causes multiple apps signed by the same developer certificate to share a user identifier. As permissions are granted to user identifiers, this causes all the apps sharing the same identifier to be granted the union of all the permissions requested by each app. To detect such leaks in an app that shares a user identifier, our system reports the exercise of an unrequested capability, which suspiciously has been requested by another app by the same author. We stress that an implicit leak requires a certain combination of apps to be installed: an app seeking to gain unauthorized capabilities can only do so if another app, with the same shared user identifier and signing key, is installed to grant the additional permission. In the context of the pre-loaded apps on the phone, we can identify whether such a colluding app exists. However, due to the fact that we cannot rule out the possibility of a colluding app being installed at a later time, its mere absence does not indicate such an implicit leak is “safe” and may not occur later.

In this work, we consider the scenario where a smartphone user has installed a third-party app on the phone. The author of the third-party app has the necessary knowledge of the phone’s...
system image, and aims to maliciously perform some high-privilege activities (e.g., recording the user’s phone conversations) through Android APIs that are protected by permission checks. To do that, the attacker chooses to not request the required permissions to elude detection or these permissions cannot be granted to third-party apps. (Examples include those permissions defined as signature or signatureOrSystem [91]). Meanwhile, we limit the attacker’s scope by assuming the Android framework (including the OS kernel) is trusted. Also, we assume that the signing key to the system image has not been leaked to the attacker. Given these constraints, a malicious app will not be able to directly access the high-privilege APIs. However, since many pre-loaded apps have the corresponding permissions, the malicious app will have gained access to a high-privilege capability if it can cause one of these apps to invoke the desired API on its behalf.

4.2.1 Explicit Capability Leak Detection

Explicit capability leaks may occur in any pre-loaded app that has requested a capability of interest in its manifest file. To detect these leaks, our system analyzes each such app in two steps. The first step, possible-path identification builds a control-flow graph to identify possible paths from a well-defined entry point (in the manifest file) to some use of the capability. After that, the second step, feasible path refinement employs field- and path-sensitive inter-procedural data flow analysis to determine which of these paths are feasible.

Possible Path Identification

Given a pre-loaded app under inspection, our system first extracts its Dalvik bytecode, and then builds a control-flow graph (CFG) to locate possible execution paths. Since constructing a CFG is a well-studied topic, we in the following focus on those Android-specific aspects that make our task complicated.

The first issue stems from indirect control-flow transfer instructions in Dalvik bytecode. Dalvik targets a hypothetical machine architecture, which does not support most forms of indirect control-flow transfer. In fact, the only indirect transfer in Dalvik’s machine language is due to the Java equivalent of pointers: object references. However, object references are rather commonly passed as arguments within an app method, and due to inheritance it is often not possible to unambiguously determine what concrete class a reference represents. During our analysis, object references will also naturally require type resolution of related objects. In our current prototype, we take a conservative approach. Specifically, when analyzing an app’s Dalvik bytecode, our system maintains a comprehensive class hierarchy. When an ambiguous reference is encountered, we consider all possible assignable classes. This is a straightforward approach, but one that will not introduce any false negatives (Section 4.5).
Another problem arises from Android’s event-driven nature. In particular, due to the large number of callbacks used by the Android framework, app execution often passes through the framework to emerge elsewhere in the app. For a concrete example, consider the `java.lang.Thread` class. This class is used to implement native threads, which Android uses in abundance to achieve better UI responsiveness. A developer can simply extend this class, implement the `run()` method, and then call the `start()` method to schedule the thread. However, if we analyze only the code contained within the app, the `run()` method does not appear to be reachable (from `start()`), despite the fact that after the `start()` method is called, control flow goes through the Dalvik VM to the underlying thread scheduler and eventually to the `run()` method. In other words, Android’s event-driven nature will unavoidably cause some discontinuity in the CFG construction if we only focus on analyzing the app code (Figure 4.2). Fortunately, beyond CFG construction, this intervening framework code is of no particular value to our analysis, and its behavior is well-defined in the Android framework APIs. Therefore, we leverage these well-defined semantics to link these two methods directly in the control flow graph, resolving the discontinuity in the process. We have applied this strategy to a number of other callbacks, such as those for message queues, timers, and GPS position updates.

Android’s use of events is so core to the platform that it is even reflected in the structure of Android apps. This leads to a final complication, because an Android app does not necessarily have only one entry point. Instead, rather than a traditional “main method” of some kind, an Android app contains one or more components defined in its manifest file. Each component can potentially define multiple entry points accessible through the Binder IPC mechanism. To take these factors into account, our prototype iterates through each entry point defined in the manifest file to build the CFG. Within each CFG, we then locate possible paths, each indicating the reachability from a known entry point to a point that exercises a specific permission of interest.
**Algorithm 2**: Capability leak detection

### Feasible Path Refinement

The previous step produces control-flow graphs which may represent a tremendous number of potential paths. Among these possible paths, not all of them lead to a dangerous call that exercises a permission of interest, and of those that do, not all are feasible. Therefore, we employ inter-procedural data flow analysis to find paths that are both feasible and result in a dangerous call.

Specifically, we use symbolic path simulation, a path-sensitive data flow analysis technique. The underlying intuition is that a path of program execution can be modeled as a set of program states, each dependent on the last. For this set of states to be feasible, each program point (or instruction) must follow from the preceding ones. Similar to other data flow analysis techniques, symbolic path simulation implements an iterative algorithm that converges on a fix-point. At each program point, the set of input states are fed through a transfer function (representing the operation performed by that instruction) to produce a set of output states. However, before these output states are used as input states for that program point’s successors, we verify that their constraints are consistent. In this way, infeasible paths are not fed forward through the analysis.
As a field- and path-sensitive symbolic simulation algorithm (summarized by Algorithm 2), our approach considers multiple similar concrete paths through a program at once, and condenses methods into parameterized summaries that relate their inputs to their outputs. Each state in the analysis encodes the value of data fields with constraints, allowing some similar states to be joined with one another. Particularly, the algorithm operates in the standard fashion for data flow analysis: a worklist is maintained of actively-considered states, and a transfer function ($\delta$) is used to generate new states from a given state. Only new states are added to the worklist, so eventually the algorithm converges on a solution that represents all the feasible states reachable from a given entry point.

By considering certain properties of the Android platform, we can optimize our algorithm in a number of aspects. For example, we accelerate the process by using method summaries to avoid recursively considering the same method-call chains multiple times. To save space, joining (rather than simply adding) new states to the worklist and visited-state list make the algorithm scale better both in terms of time and memory. Our implementation recognizes the value constraints placed on each memory item, and will aggressively merge similar states where possible. As an example, if two states are joined that only differ by whether a boolean value is true or false, the resulting state will simply remove any constraint on the boolean value. In this way, fewer states need to be remembered, and fewer successors calculated using the transfer function $\delta$.

Moreover, since an Android app can define multiple entry points, there is a need to produce a separate set of potential paths for each. These paths do not include any executed instructions in the app prior to the entry point, which excludes such code as any constructors that set the initial state of the app. Due to the fact that the entry points in an app can be invoked in any sequence, we opt to take a conservative approach by assuming that a field might contain any assignable value. As that field is used along a path of execution, the list of possible values shrinks each time it is used in a way that renders some candidate values impossible. When reducing infeasible paths, we also face the same type inference problem experienced in the first step. Fortunately, the set of inferences built up by symbolic path simulation naturally mitigates the path explosion caused by our first step. Specifically, object instances can be tracked during the path simulation, and some paths will become infeasible after the system infers the object’s type somewhere along a path. Certain Dalvik bytecode operations, especially type-carrying instructions, can also greatly help. For instance, the check-cast opcode establishes that its operand can be assigned to the supplied type, or an exception is thrown.

In addition, the execution of our algorithm also involves handling Android framework APIs or methods that do not belong to the app. Specifically, the app under inspection may invoke certain APIs that are exported by the Android framework. In our algorithm, the transfer function for a method invocation opcode is a method summary, which essentially phrasers the method’s
outputs in terms of its inputs. Without statically analyzing the code for an external method – and all of its dependencies – we cannot build such a summary. Yet analyzing the entire Android framework would easily lead to state explosion and scalability issue. To address that, we again leverage the well-defined API semantics of the Android framework. Specifically, it contains a remarkably robust set of predefined libraries, which reduces the need for developers to pull in third-party libraries to support their code. By summarizing these built-in classes ahead of time, we can avoid paying the time, space, and complexity costs associated with doing so each time during application analysis. In our prototype, we found that this approach allows us to phrase some functions more succinctly than the algorithm would, as we can trim out unimportant details from the summaries.

During this infeasible path pruning step, we also need to account for explicit permission checks within the identified path. An app might allow any caller to invoke its entry points, yet deny unprivileged callers access to dangerous functionality by explicitly checking the caller’s credentials before any dangerous invocations. Such an arrangement would not constitute a capability leak, and so should not be reported. A naïve solution would be to mark any path encountering an interesting permission check as infeasible. However, our approach does not know what kind of dangerous call lies at the end of the path beforehand. Allowing unrelated permission checks to mark whole paths as infeasible would therefore introduce false negatives. Instead, we model the permission system within our artificial method summaries. Explicit permission checks set a flag along their “true” branch; if that path of execution later encounters a corresponding dangerous call, it is not reported as a capability leak.

A side benefit of performing this kind of analysis is that it models all data flow assignments, not just those relating to branch conditions. As a result, we can trace the provenance of any arguments to the dangerous method. With such information, we can characterize the severity of the capability leak. A capability leak that directly passes through arguments from the external caller is obviously worse than one that only allows invocation with constant values, and this design can distinguish between the two. Given that path feasibility is undecidable, our design errs on the side of caution: it will not claim a feasible path is infeasible, but might claim the reverse is true. As a result, this argument information is valuable, as it can be used to generate a concrete test case that verifies the detected capability leak.

### 4.2.2 Implicit Capability Leak Detection

When detecting explicit capability leaks, we focus on those apps that request permissions of interest in their manifest files. If an app has a `sharedUserId` in its manifest but does *not* request a certain (dangerous) permission, we also need to investigate the possibility of an implicit capability leak.
To detect implicit capability leaks, we employ a similar algorithm as for explicit leaks with necessary changes to reflect a fundamental difference in focus. Specifically, explicit capability leak detection assumes the caller of an app’s exposed API is malicious, while implicit capability leak detection assumes the app itself might be malicious. Accordingly, instead of only starting from the well-defined entry points in the explicit leak detection, there is a need to broaden our search to include the app’s initialization.

Unfortunately, modeling the initialization process in an Android app is somewhat complicated. Specifically, there are two kinds of constructors to handle: (1) Instance constructors that are explicitly invoked in the Dalvik bytecode with the `new-instance` bytecode operation and (2) Class constructors or static initialization blocks that are implicitly invoked the first time a class is used. Accordingly, instance constructors are relatively straightforward to handle as they need to be explicitly invoked. However, class constructors are more complicated. In particular, a class constructor may be invoked in a number of scenarios: it is instantiated with the `new` keyword, a static member of the class is referenced, or one of its subclasses is likewise initialized. This means that this type of initialization can occur in a variety of orders. In our prototype, we treat all of the relevant instructions as branches, and take into account the class loading order to determine the path feasibility. Also, in our system, we consider a capability to have been implicitly leaked if there is any way to exercise it, which is different from explicit capability leak detection. (This has implications in changing method summaries used for pruning infeasible paths – Section 4.2.1.)

Finally, once we have identified that an implicit capability leak exists, we can perform an additional step to determine whether that leak may actually be exercised. In the context of a phone’s system image, we can determine the runtime permissions granted to each shared user identifier by crawling the manifest files of all the packages in the image. We union the permissions granted to each application with a given shared user identifier, which yields the set of permissions given to each of them. We report any implicitly leaked permissions contained within that set.

### 4.3 Implementation

We have implemented a Woodpecker prototype that consists of a mixture of Java code, shell scripts and Python scripts. Specifically, our static analysis code was developed from the open-source baksmali disassembler tool (1.2.6). We could have developed Woodpecker as a set of extensions to an existing Java bytecode analysis tool (e.g., Soot [26] or WALA [28]). Given concerns over the accuracy of existing Dalvik-to-Java bytecode translators, we opted to operate directly on baksmali’s intermediate representation. To detect possible capability leaks in an Android phone, our system first leverages the Android Debug Bridge (adb) tool [2] to obtain
access the phone’s system image, mainly those files in the /system/app and /system/framework directories. These directories contain all of the pre-installed apps on the device, as well as any dependencies they need to run.

After obtaining the phone image, we then enumerate all pre-installed apps. For each app, our system decompresses the related Android package (apk) file to extract its manifest file (AndroidManifest.xml) and then pairs it with the app’s bytecode (either classes.dex or its odex variant). A standalone script has been developed to extract all the pre-installed apps and disassemble them to extract their bytecode for subsequent analysis. Depending on the number of apps installed on the device and the complexity or functionality implemented in these apps, this process typically takes on the order of ten minutes per smartphone image.

After extracting the app manifest files, we further comb through them for two things: requests for any permissions of interest and the optional sharedUserId attribute. Apps that are granted related permissions are checked for explicit capability leaks, while those with the sharedUserId attribute set are checked for implicit capability leaks. Naturally, we also compute the actual set of permissions granted to each pre-loaded app by combining all the permission requests made with the same sharedUserId.

4.3.1 Control-Flow Graph Construction

We iterate through each selected pre-loaded app to detect possible capability leaks. As there are tens of dangerous permissions defined in the Android framework, instead of building a specific control-flow graph (CFG) for each permission, we choose to first build a generic CFG to assist our static analysis.

In particular, we start from each entry point and build the respective CFG. The generic whole-program CFG will be the union of these CFGs. There is some subtlety in Android involved in mapping the components defined in the manifest file to their actual entry points. Some entry points are standard and can be readily determined by the type of components contained within the app. Specifically, there are in total four types, and each has a predefined interface to the rest of the system. For instance, any “receiver” defined in the manifest file must subclass android.content.BroadcastReceiver. In such cases, inspecting the class hierarchy allows to determine that the “onReceive(Context, Intent)” method is an entry point (as per the specification).

Moreover, among these four types, three of them solely take data objects as inputs through their entry points, but services can be different. In particular, Android defines a CORBA-like binding language, the Android Interface Definition Language (AIDL), which allows services to expose arbitrary methods to other apps. aidl files are used at compile-time to manufacture Binder stubs and skeletons that encapsulate the necessary IPC functionality. At run-time, the
component’s `onBind(Intent)` method is called by the system, which returns an `android.os.-Binder` object. The methods contained within this object are then exported to callers that have a compatible skeleton class. Since we only analyze the bytecode and do not have access to the original `aidl` files used to define the interface, there is a need to further parse and infer the internal structure of the `Binder` object. Each such object contains an `onTransact()` method that is passed a parcel of data that encodes which method to call. We can then treat this method as an entry point in order to build our CFG. However, once the graph has been built, it is more semantically accurate to treat the embedded method calls in `onTransact()` as entry points for the purposes of our feasible path refinement stage.

From another perspective, Android apps essentially expose a set of callbacks to the system instead of a single “main method.” Our system leverages the knowledge of how these callbacks are defined in Android to identify them. In addition, the Android framework defines many other callbacks at run-time, which will similarly cause discontinuities in the CFG generation. One example is the previous `Thread.start() → run()` scenario. In our prototype, instead of statically analyzing the entire Android framework, we opt to use knowledge of the framework’s semantics to connect the registration of a callback to the callback itself. To automate this process, we provide a boilerplate file that represents knowledge about the framework. This file contains simplified definitions for any explicitly-modelled method in the framework, written in the `dex` format; it is fed into our system alongside the app’s code to facilitate CFG construction.

### 4.3.2 Capability Leak Detection

With the constructed CFG and the set of entry points, we then aim to identify possible execution paths from one of the entry points to some use of an Android API that exercises a permission of interest. If the path is not protected by the appropriate permission checks and its entry point is publicly accessible, an explicit capability leak is detected. Due to the large number of sensitive permissions defined in the Android framework, our study chooses thirteen representative permissions marked `dangerous`, `signature` or `signatureOrSystem`. These permissions are summarized in Table 4.1 and were chosen based on their potential for abuse or damage. For example, the `SEND_SMS` permission is a favorite of malware authors [92]: it can be used to send messages to costly premium numbers, which pay the culprit for each such text.

For each chosen permission, our first step is to identify the list of related Android APIs that might exercise the permission. However, such a list is not easy to come by. In fact, we found out that though Android’s permission-based security model might be comprehensive enough in specifying the permissions required to access sensitive data or features, the available API documentation is incomplete about which APIs a permission grants access to. Specifically, when dealing with various apps in the system image, we encountered numerous permissions...
Table 4.1: The list of 13 representative permissions in our study (†: we omit the common android.permission. prefix)

<table>
<thead>
<tr>
<th>Permission†</th>
<th>Capability</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS_COARSE_LOCATION</td>
<td>Access coarse location (e.g., WiFi)</td>
</tr>
<tr>
<td>ACCESS_FINE_LOCATION</td>
<td>Access fine location (e.g., GPS)</td>
</tr>
<tr>
<td>CALL_PHONE</td>
<td>Initiate a phone call (without popping up an UI for confirmation.)</td>
</tr>
<tr>
<td>CALL_PRIVILEGED</td>
<td>Similar to CALL_PHONE, but can dial emergency phone numbers (e.g., 911)</td>
</tr>
<tr>
<td>CAMERA</td>
<td>Access the camera device</td>
</tr>
<tr>
<td>DELETE_PACKAGES</td>
<td>Delete existing apps</td>
</tr>
<tr>
<td>INSTALL_PACKAGES</td>
<td>Install new apps</td>
</tr>
<tr>
<td>MASTER_CLEAR</td>
<td>Remove user data with a factory reset</td>
</tr>
<tr>
<td>READ_PHONE_STATE</td>
<td>Read phone-identifying info. (e.g., IMEI)</td>
</tr>
<tr>
<td>REBOOT</td>
<td>Reboot the device</td>
</tr>
<tr>
<td>RECORD_AUDIO</td>
<td>Access microphones</td>
</tr>
<tr>
<td>SEND_SMS</td>
<td>Send SMS messages</td>
</tr>
<tr>
<td>SHUTDOWN</td>
<td>Power off the device</td>
</tr>
</tbody>
</table>

not meant for general consumption – and that therefore do not even have formally specified APIs. One example is “android.permission.MASTER_CLEAR,” which allows an app to perform a factory reset of the smartphone. This permission is marked as signatureOrSystem, so only apps included in the system image can request it; it is intended to be implemented by the vendor and only used by the vendor, so none of the APIs listed in the API documentation check this permission.

For each related permission and the associated Android APIs, our next step then reduces the generic CFG to a permission-specific CFG. Within the reduced CFG, we can then apply the Algorithm 2 (in Appendix A) to locate possible execution paths from an entry point to the associated Android APIs. For each identified path, we further look for the presence of certain permission checks. Our experience indicates that some permission checks are already defined in the manifest file (and thus automatically enforced by the Android framework). However, many others will explicitly check their caller’s permissions. In our prototype, we resort to the Android Open Source Project (AOSP) to find explicit permission checks in the framework. There are also some cases that do not fall under the AOSP. For them we have to apply baksmali to representative phone images and then manually examine each explicit permission check. Using the previous example of “android.permission.MASTER_CLEAR,” Android provides an interface, android.os.ICheckinService that declares the masterClear() method. The Samsung Epic 4G’s factory reset implementation contains a class com.android.server.FallbackCheckinService. This class implements this Android interface, whose masterClear() method explicitly checks the “android.permission.MASTER_CLEAR” permission.

To facilitate our static analysis, our prototype also includes a fictitious dangerous class that has many static permission-associated member fields. Each identified Android API call, if
Table 4.2: Eight studied Android smartphones

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Model</th>
<th>Android Version</th>
<th># Apps</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTC</td>
<td>Legend</td>
<td>2.1-update1</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>EVO 4G</td>
<td>2.2.2</td>
<td>160</td>
</tr>
<tr>
<td></td>
<td>Wildfire S</td>
<td>2.3.2</td>
<td>144</td>
</tr>
<tr>
<td>Motorola</td>
<td>Droid</td>
<td>2.2.2</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td>Droid X</td>
<td>2.2.1</td>
<td>161</td>
</tr>
<tr>
<td>Samsung</td>
<td>Epic 4G</td>
<td>2.1-update1</td>
<td>138</td>
</tr>
<tr>
<td>Google</td>
<td>Nexus One</td>
<td>2.3.3</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td>Nexus S</td>
<td>2.3.3</td>
<td>72</td>
</tr>
</tbody>
</table>

present in an execution path being analyzed, will update the member field related to the associated permission. As a result, we can detect dangerous calls by simply listing the related member fields in this class. Similarly, to model the impact a caller’s permissions have on whether a dangerous call can succeed, we use another fictitious permission class. This class contains a number of member fields and an artificial method definition for Context.checkCallingPermission(). This method sets these member fields dependent upon the permission it is called with. In other words, each member field flags whether a path of execution has checked a particular permission. During an explicit capability leak analysis run, we only consider a capability to have been leaked if a state exists that contains a dangerous-call field modification (maintained in dangerous class) and does not have the corresponding permission-check flag set (in permission class). Implicit capability leak analysis does not need to be concerned about the value of the permission-check flags. Instead, it is sufficient to have a dangerous call field modification (in dangerous class).

4.4 Evaluation

In this section, we present the evaluation results of applying Woodpecker to eight smartphones from four vendors, including several flagship phones billed as having significant additional bundled functionality on top of the standard Android platform. We describe our methodology and tabulate our results in Section 4.4.1. In Section 4.4.2, we present a case study for each type of capability leak, explicit and implicit. Lastly, Section 4.4.3 consists of a performance measurement of our system, both in terms of the accuracy of its path-pruning algorithm and its speed.
Table 4.3: Capability leak results of eight Android-based smartphones (E: explicit leaks; I: implicit leaks)

<table>
<thead>
<tr>
<th>Permission</th>
<th>HTC</th>
<th>EVO 4G</th>
<th>Wildfire S</th>
<th>Motorla</th>
<th>Droid X</th>
<th>Samsung</th>
<th>Google</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS_COARSE_LOCATION</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ACCESS_FINE_LOCATION</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CALL_PRIVILEGED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CAMERA</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DELETE_PACKAGES</td>
<td>✓²</td>
<td>✓²</td>
<td>✓²</td>
<td>✓²</td>
<td>✓²</td>
<td>✓²</td>
<td>✓²</td>
</tr>
<tr>
<td>INSTALL_PACKAGES</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MASTER_CLEAR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>READ_PHONE_STATE</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>REBOOT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RECORD_AUDIO</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SEND_SMS</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHUTDOWN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>6</td>
<td>2</td>
<td>8</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

4.4.1 Results Overview

In order to assess capability leaks posed in the wild, we selected phones representing a variety of manufacturers and feature sets. Table 4.2 shows the phone images and their versions we analyzed using Woodpecker. These phones span most of the 2.x version space, and as shown by the app count for each phone image, some are considerably more complex than others.

Running Woodpecker on each phone image produces a set of reported capability leak paths. For each reported path, we then manually verify the leak by tracing the path through the disassembled Dalvik bytecode. For explicit capability leaks whose paths seem plausible, we then craft a test application and run it on the actual device, where possible. The results are summarized in Table 4.3.

After identifying these capability leaks, we spent a considerable amount of time on reporting them to the corresponding vendors. As of this writing, Motorola and Google have confirmed all the reported vulnerabilities in the affected phones. HTC and Samsung have been really slow in responding to, if not ignoring, our reports/inquiries. Though the uncovered capabilities leaks on the HTC and Samsung phones have not been confirmed by their respective vendors, we have developed a test app to exercise and confirm all the discovered (explicit) capability leaks on the affected phones.

We believe these results demonstrate that capability leaks constitute a tangible security weakness for many Android smartphones in the market today. Particularly, smartphones with more pre-loaded apps tend to be more likely to have explicit capability leaks. The reference implementations from Google (i.e., the Nexus One and Nexus S) are rather clean and free from capability leaks, with only a single explicit leak (marked as ✓² in Table 4.3) due to an app com.svox.pico. This app defines a receiver, which can be tricked to remove another app,
Our data also show that these capability leaks are not evenly distributed among smartphones – at least for the 13 permissions we modelled. For example, those smartphones with system images (i.e., the Motorola Droid) close to the reference Android design (i.e., the Nexus One and Nexus S) seem to be largely free of capability leaks, while some of the other flagship devices have several. Despite this general trend, we caution against drawing any overly broad conclusions, as some devices (e.g., the Motorola Droid X) with higher app counts nevertheless contained fewer capability leaks than substantially simpler smartphones (e.g., the HTC Legend).

4.4.2 Case Studies

To understand the nature of capability leaks and demonstrate the effectiveness of our system, we examine three scenarios in depth. These scenarios were selected to illustrate some of the patterns we encountered in practice, as well as how our system was able to handle them.

Explicit Capability Leaks (Without Arguments)

The simplest scenario, from Woodpecker’s perspective, involves an entry point calling a dangerous capability that is not influenced by any arguments. These capabilities tend to have simpler control flows, as there are no arguments to validate or parse. The Samsung Epic 4G’s `MASTER_CLEAR` explicit capability leak is of this type, which once exploited, allows for unauthorized wiping of user data on the phone.

To understand how Woodpecker detects this explicit capability leak, we first explain the normal sequences when the `MASTER_CLEAR` capability is invoked. Specifically, the Samsung Epic 4G’s phone image has a pre-loaded app, `com.sec.android.app.SelectiveReset`, whose purpose is to display a confirmation screen that asks the user whether to reset the phone. The normal chain of events has another system app broadcast the custom `android.intent.action.SELECTIVE_RESET` Intent, which the `SelectiveResetReceiver` class (defined in the pre-loaded app) listens for. When this class receives such an intent, it opens the user interface screen (`SelectiveResetApp`) and waits for the user to confirm their intentions. Once this is done, the `SelectiveResetService` is started, which eventually broadcasts an intent `android.intent.action.SELECTIVE_RESET_DONE`. The original `SelectiveResetReceiver` class listens for this Intent and then calls `CheckinService.masterClear()`.

Our system detects the last part of the above chain starting after the broadcasted intent `android.intent.action.SELECTIVE_RESET_DONE` is received in the same pre-loaded app.

---

3This `com.svox.pico` app implements a text-to-speech engine that the accessibility APIs use to talk. However, it exports a public receiver interface, `com.svox.pico.LangPackUninstaller` for `android.speech.tts.engine.TTS_DATA_INSTALLED` intents. If such an intent is received, this app will blindly remove another app, `com.svox.langpack.installer`, whose name is hard-coded in the implementation.
In particular, the intent arrives at one entry point defined in the app’s manifest file (i.e., the `onReceive(Context, Intent)` method within `SelectiveResetReceiver`), which then executes a rather straightforward `Intent`-handling code sequence: (1) determines that the received `Intent` is an `android.intent.action.SELECTIVE_RESET_DONE` operation; (2) gets the `CheckinService` that contains the master clear functionality; (3) checks whether it was retrieved successfully; and (4) calls `CheckinService.masterClear()` in a worker thread. Since `CheckinService.masterClear()` takes no arguments, no additional dataflow analysis needs be performed to characterize the capability leak.

In our experiments, we also found other capability leaks of the same nature, including the `REBOOT` and `SHUTDOWN` leaks on the HTC EVO 4G. On the same phone, we also found a new vendor-defined capability `FREEZE` exposed by a system app, which disables the phone’s touchscreen and buttons until the battery is removed. In those cases, there is literally no control flow involved, making these capability leaks trivial to exploit. We point out that analyzing explicit capability leaks that involve arguments works in much the same fashion. Regardless, another explicit capability leak case study is included (Section 4.4.2) that accounts for the presence of arguments.

**Explicit Capability Leak (With Arguments)**

Looking beyond simple imperative capability leaks, we consider more complicated cases that involve argument-taking capabilities. For example, Android’s SMS API consists of three methods, each of which takes five or six arguments. The HTC phones have an explicit leak of this capability that entails significant preprocessing of these arguments, which we examine as an additional case study to illustrate how our system works.

On these phones, the general `com.android.mms` messaging app has been extended to include a non-standard service, `com.htc.messaging.service.SmsSenderService`, which is used by other vendor apps to simplify sending SMS messages. This service can be started with an `Intent` that contains a number of additional data key-value pairs, known as `Extras`. Each `Extra` contains some data about the SMS to be sent, such as the message’s text, its call-back phone number, the destination phone number, and so on.

The `SmsSenderService` service (Figure 4.3) processes these fields in its `onStart(Intent, int)` entry point, ensuring that the mandatory key-value pairs exist, including the message body and destination phone number. If they do, the `Intent` is bundled into a `Message` and sent to the `SmsSenderService$ServiceHandler` class via the Android message-handling interface. This interface is designed to allow different threads of execution to communicate using a queue of `Messages`. The typical paradigm uses a subclass of `android.os.Handler` to poll for new `Message` objects, using a `handleMessage(Message)` method. Such `android.os.Handler` objects also expose methods to insert `Messages` into their queue, such as `sendMessage(Message)`.
When building possible paths and pruning infeasible paths, our system will diligently resolve the super- and sub-class relationships that bracket the message-passing code. In this case, the initial `SmsSenderService$ServiceHandler.sendMessage(Message)` call fully specifies the class that `sendMessage(Message)` will be called upon, but `SmsSenderService$ServiceHandler` does not contain a definition for that method. Looking to its superclass, `android.os.Handler`, Woodpecker finds an artificial method definition of the appropriate signature. This definition in turn calls the `android.os.Handler.handleMessage(Message)` method, which is extended by the `SmsSenderService$ServiceHandler` class. In this case, our design has no difficulty resolving these relationships, because the first call fully specifies the `SmsSenderService$ServiceHandler` class. This type information is then carried forward through the call chain as a constraint on the arguments to each call, as a class’ methods are associated with an object instantiating that class via an implicit argument (the `this` keyword).

Ultimately, the app execution flow will reach `SmsManager.sendMultipartTextMessage()`, a method that exercises the dangerous `SEND_SMS` permission. The arguments by this point have been transformed: the destination address remains the same, but the call-back number may not have been provided by the `Intent`’s data, and the message body might have been chunked into
SMS-sized pieces if it is too long. When processing this execution path, Woodpecker reports this path as feasible and thus exposing the exercised permission \texttt{SEND\_SMS}. Since the exercised capability took a number of arguments, our system also reports the provenance of each related argument to the Android API, which allows for straightforwardly linking the API arguments back to the original \texttt{Intent} passed to the entry point at the very beginning. In other words, by simply including a premium number in the intent, the built-in app will start sending SMS messages to this premium number!

Our experience indicates most capability leaks we detected are of this form. For example, the explicit leak of \texttt{CALL\_PHONE} capability in Samsung Epic 4G involves passing a component a “technical assistance” phone number, which it calls after considerable processing. Similarly, all the tested HTC phones export the \texttt{RECORD\_AUDIO} permission, which allows any untrusted app to specify which file to write recorded audio to without asking for the \texttt{RECORD\_AUDIO} permission.

\textbf{Implicit CapabilityLeaks}

Explicit leaks seriously undermine the permission-based security model of Android. Implicit leaks from another perspective misrepresent the capability requested by an app. In the following, we choose one representative implicit leak and explain in more detail. Specifically, the HTC Wildfire S has a built-in MessageTab app, \texttt{com.android.MessageTab}, which uses the \texttt{CALL\_PRIVILEGED} capability (marked as ✓ in Table 4.3) without declaring it in its manifest. This MessageTab app is intended to manage the phone’s SMS messages, allowing the user to review sent messages and send new ones. For the sake of convenience, this app links messages sent to contacts with the appropriate contact information, allowing the user to dial contacts directly through a “contact details” screen. However, this app does not declare the correct permissions to call phone numbers, as it only requests SMS-related permissions: neither the \texttt{CALL\_PHONE} nor \texttt{CALL\_PRIVILEGED} permission occur in its manifest. On the other hand, MessageTab does declare a \texttt{sharedUserId} attribute: “\texttt{android.uid.shared}.” This user identifier is used by a number of core Android apps, including \texttt{com.android.htcdialer} – which has both phone-dialing permissions.

When analyzing this app, Woodpecker reports an implicit leak in the \texttt{com.android.MessageTab.ContactDetailMessageActivity2} activity component. Specifically, this component has a \texttt{onResume()} method – an entry point called when the activity is displayed on the screen. In this case, it is used to instruct on how to build a list of contacts to display on the screen, by calling \texttt{com.htc.widget.HtcListView.onCreateContextMenu()} with a callback object (\texttt{ContactDetailMessageActivity2$3}). When the user long-presses one of these contacts, that callback object’s \texttt{onCreateContextMenu()} method is called. This method then calls \texttt{ContactDetailMessageActivity2.addCallAndContactMenuItems()} to make the contacts’ context menus. A call to a helper method, \texttt{android.-}
mms.ui.MessageUtils.getMakeCallDirectlyIntent(), builds the Intent to send to dial a contact. This helper method builds the actual android.intent.action.CALL_PRIVILEGED Intent, which will be broadcasted when the user clicks on the contact. From the disassembled code, the addCallAndContactMenuItems() method also registers an ContactDetailMessageActivity2$MsgListMenuClickListener object as a callback for the click-able contact. This object’s onMenuItemClick(MenuItem) method is then called, which takes the Intent associated with the contact and calls com.android.internal.telephony.ITelephony.dialWithoutDelay(Intent) with it, which immediately dials a phone number.

Note that this implicit capability leak traversed a number of callbacks that either require user intervention or are very visible to the user. These callbacks would normally not be considered useful for an explicit capability leak, which assumes a malicious caller. However, as implicit capability leaks assume that the app itself may be malicious, our algorithm simply reports them by not making such value judgments when considering possible execution paths.

4.4.3 Performance Measurement

Next, we evaluate the performance of our prototype, in terms of both the effectiveness of its path pruning algorithm and the amount of time it takes to process a smartphone’s system image.

To measure how well Woodpecker’s path pruning algorithm eliminates infeasible paths, we consider its output from the experiments with a single permission, android.permission.SEND_SMS. In particular, we run only the possible-paths portion of the algorithm (i.e., with no pruning) and identify how many paths may possibly leak a dangerous capability. Our results show that for each phone, Woodpecker will report more than 8K possible paths. This surprisingly large number is due to the conservative approach we have taken in resolving an ambiguous reference to assignable classes. Fortunately, our re-run of the full system by pruning the infeasible paths immediately brings the number to the single digits. Specifically, our system only reports capability leaks in the HTC phones, especially 2, 3, 2 for the HTC Legend, EVO 4G, and Wildfire S respectively. Among the reported leaks, we then manually verify the correctness of the pruned paths. The results show they are all valid with no false positives. Note that the presence of one single path is sufficient to leak the related capability. We do not measure false negatives due to the lack of ground truth in the tested phone images. However, because of the conservative approach we have been taking in our prototype, we are confident in its low false negatives.

For the processing time, we measure them directly by running our system multiple times over the tested smartphone images. We analyze each image ten times on an AMD Athlon 64 X2 5200+ machine with 2GB of memory and a Hitachi HDP72502720 7200 rpm hard drive. The mean of these results are summarized in Table 5.4. Each phone image took at most a little over
Table 4.4: Processing time of examined smartphone images

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Model</th>
<th>Processing Time</th>
<th># Apps</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTC</td>
<td>Legend</td>
<td>3366.63s</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>EVO 4G</td>
<td>4175.03s</td>
<td>160</td>
</tr>
<tr>
<td></td>
<td>Wildfire S</td>
<td>3894.37s</td>
<td>144</td>
</tr>
<tr>
<td>Motorola</td>
<td>Droid</td>
<td>2138.38s</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td>Droid X</td>
<td>3311.94s</td>
<td>161</td>
</tr>
<tr>
<td>Samsung</td>
<td>Epic 4G</td>
<td>3732.56s</td>
<td>138</td>
</tr>
<tr>
<td>Google</td>
<td>Nexus One</td>
<td>2059.47s</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td>Nexus S</td>
<td>1815.71s</td>
<td>72</td>
</tr>
</tbody>
</table>

an hour to process. We believe the average time (≈ 51.0 minutes) per image to be reasonable given the offline nature of our tool, which has not yet been optimized for speed.

4.5 Discussion

Our system has so far uncovered a number of serious capability leaks in current smartphones from leading manufacturers. Given this, it is important to examine possible root causes and explore future defenses.

First of all, capability leaks essentially reflect the classic confused deputy attack [104] where one app is tricked by another into improperly exercising its privileges. Though one may easily blame the manufacturers for developing and/or including these vulnerable apps on the phone firmware, there is no need to exaggerate their negligence. Specifically, the permission-based security model in Android is a capability model that can be enhanced to mitigate these capability leaks. One challenge however is to maintain the integrity of those capabilities when they are being shared or opened to other unrelated apps. In other words, either the capability-leaking app needs to ensure that it will not accidentally expose its capability without checking the calling app’s permission, or the underlying Android framework needs to diligently mediate app interactions so that they do not inappropriately violate the integrity of a capability. However, such inter-app interactions are usually application-specific, so it is hard for the Android framework to infer the associated semantics.

Second, to avoid unsafely exposing capabilities, we can also develop a validator tool and release it together with the Android SDK. Note that such a validator tool needs to handle the various ways an app can interact with the Android permission model. Specifically, Android uses string identifiers to represent permissions, and permission information can be encoded in either the app’s manifest or code, which indicates that the permission model cannot be considered
type-safe. Accordingly, conventional Java source code analysis tools are not aware of the impact permissions have on program execution.

Woodpecker represents our first step towards such a validator tool for capability leak detection. Though it has identified serious capability leaks in current Android phones, it still has a number of limitations that need to be addressed. For example, other than tightening the underlying implementation and incorporating latest development of accurate, scalable points-to analysis [60, 161, 162], our current prototype now handles only Dalvik bytecode and needs to be extended to accommodate native code. In doing so, the issue of dynamically loaded code would be raised, which is a limitation for purely static approaches. Also, our current prototype only handles 13 permissions that are defined by the framework itself. However, many more exist, and apps are free to define new ones. In a follow-on work, SEFA [179], we show that a refined system that handled more predefined permissions produced similar results for them, confirming the extent of the problem. However, adding support for app-defined permissions would lead to another class of capability leaks: chained capability leaks. To illustrate, consider three apps: A, B, and C. C has the CALL_PHONE capability, which it safely exposes to B by defining a new MY_CALL_PHONE permission. This new permission is acquired by B. For a chained leak to occur, B opens up the new permission unsafely to A. As a result, there is a call chain A→B→C, which could leak the CALL_PHONE capability. Since the new permission MY_CALL_PHONE can be arbitrary and specific to a particular implementation, we need to explore innovative ways to extend our prototype to accommodate such chained capability leaks.

Finally, our study only examines capability leaks among pre-loaded apps in the phone firmware. We also expect the leaks could occur among third-party user apps. Note that phone images are relatively homogeneous and static with usually a somewhat infrequent update schedule. Capability leaks, especially explicit ones, on phone images are of great interest to malicious third parties. Implicit leaks, on the other hand, appear to be relatively rare, which we assume are more software engineering defects than a real security threat. However, for third-party apps, implicit leaks could constitute collusion attacks that directly undermine the app market model. Specifically, app markets do not report the actual permissions granted to an app. Instead they report only the permissions an app requests or embodied in the manifest file. As a result, a cohort of seemingly innocuous apps could conspire together to perform malicious activities and the user may not be informed of the true scope of their permissions within the system. Meanwhile, we hypothesize that explicit leaks in user-installed apps may be less common and useful, as an app must have both a sizable installed base and unwittingly expose some interesting functionality in order for an attacker to derive much benefit from exploiting the leaked capabilities.
4.6 Summary

In this chapter, we present a system called Woodpecker to examine how the Android-essential permission-based security model is enforced on current leading Android-based smartphones. In particular, Woodpecker employs inter-procedural data flow analysis techniques to systematically expose possible capability leaks where an untrusted app can obtain unauthorized access to sensitive data or privileged actions. The results are worrisome: among the 13 privileged permissions examined in our survey, 11 were leaked, with individual phones leaking up to eight permissions. These leaked capabilities can be exploited to wipe out the user data, send out SMS messages (e.g., to premium numbers), record user’s conversations, or obtain the user’s geo-location data on the affected phones – all without asking for any permission.

We attribute the uneven enforcement of this new security model to the customization process within Android, which does not provide strong tooling to ensure security assumptions do not break as vendors attempt to add functionality to the base platform. This leads to the aforementioned improper architectural organizations that lead to privilege reexposure, supporting our thesis.
Chapter 5

Sub-Application-Level Capability Leaks

5.1 Background

As mentioned in Chapter 1, smartphones are defined by the ability to download and run third-party apps that provide additional useful features. In other words, instead of being restricted to the functions provided by phone manufacturers, carriers, or limited affiliates, smartphone users can partake of thousands of apps designed for purposes unforeseen by the parties involved in making and distributing the devices. Furthermore, platform vendors (e.g., Google and Apple) also provide centralized app markets where users can simply tap through the process of browsing, searching, purchasing, downloading, and installing these apps.

As part of the mobile ecosystem, app developers, largely motivated by financial incentives, submit their apps to centralized app markets for users to access. Notice that on the Android platform, almost two-thirds of all apps are free to download [14]. To be compensated for their work, many app developers incorporate an advertisement library (also known as an ad library) in their apps. At run-time, the ad library communicates with the ad network’s servers to request ads for display and might additionally send analytics information about the users of the app. (For simplicity, we use the term ad libraries to represent both ad libraries and analytics libraries.) The ad network then pays the developer on an ongoing basis, based on metrics that measure how much exposure each individual app gives to the network and its advertisers.

Unfortunately, on Android, permissions cannot be assigned to any entity smaller than an individual app. There is no facility to grant different components of an app different amounts of access to phone features, nor is there any unified mechanism to describe to the user how that access will be used within an app. As a result, in-app ad libraries often represent internal capability leaks. To elaborate, at install time, the user is only asked to grant the app’s author
access to certain phone features. Subsequently, and unbeknownst to the user, the author then
degrees that authority to unknown third parties, constituting a capability leak. This lack of
transparency is endemic in the ad-supported app world, which ultimately led the FTC to issue
privacy disclosure guidelines [21].

In this chapter, we aim to study existing in-app ad libraries and evaluate potential risks from
them. Specifically, we focus on the Android platform and determine what risks the popular ad
libraries on Android may pose to user’s privacy and security. To this end, we collected 100,000
apps from Google Play over a three-month period, i.e., March-May, 2011. Among these apps,
we identify and extract 100 representative ad libraries that are used in 52,067 (or 52.1%) of
them. To facilitate our analysis, we further developed a static analysis tool called AdRisk to
analyze the extracted ad libraries and report possible risks. In particular, our current analysis
mainly focuses on those “dangerous” permissions (Section 5.2) defined in the standard Android
framework, seeking to identify their possible (mis)use by ad libraries.

Our analysis revealed a number of privacy and security issues in the 100 representative ad
libraries. In particular, most ad libraries collect private information. While some of them may
use these information for legitimate purposes (i.e., the user’s location for targeted advertising),
we noticed a few ad libraries invasively collect information, such as the user’s call logs, account
information or phone number. Such information can be used to deduce the true identity of the
user, enabling more comprehensive tracking of the user’s habits – at the cost of all pretense
of privacy. One particular popular ad library (used in 4190 apps in our dataset) even allows a
variety of personal information to be directly accessible to the advertisers, creating unnecessary
additional opportunities for misuse. We also found out that some ad libraries will download
additional code at runtime from remote servers and execute it in the context of the running
app, opening up the opportunities for exploitation and abuse and making it impossible to ensure
its integrity. In fact, we have confirmed one particular case that fetches and loads suspicious
payloads. After this discovery, we reported the infected apps (7 in our dataset) to Google; all of
these apps were swiftly removed from Google Play. These results call for the need for additional
mechanisms to regulate the behavior of ad libraries on Android.

The rest of this chapter is organized as follows: Section 5.2 explains how ad libraries are
embedded in host apps, covering relevant portions of the Android framework in the process. Sec-
tion 5.3 describes the system design to assess the threat posed by ad libraries, while Section 5.4
contains the implementation and evaluation results. Section 5.5 considers the implications and
limitations of our work. Lastly, we summarize our findings in Section 5.6.
Figure 5.1: The (abbreviated) AndroidManifest.xml file in the popular Angry Birds Android app (com.rovio.angrybirdsrrio)

5.2 Anatomy of an Ad-Supported App

To understand how an ad library is embedded into an Android app, we will consider a popular app, i.e., Rovio’s Angry Birds, as an example. Initially a paid iPhone app, Angry Birds moved to an ad-supported model when Rovio ported it to Android. The game is free to download, but ads are displayed periodically during play and while loading new levels; these ads generate $1 million a month in revenue for Rovio.

Since Rovio is not in the advertising business, the company turned to third-party advertising networks to monetize Angry Birds on Android. This is a common arrangement and natural choice for smartphone app developers. After registering some financial information with an ad network, developers receive a developer identifier and a SDK. The SDK’s documentation includes instructions on how to use the included ad library. Ad libraries are designed to be embedded in the app that uses them, so the instructions include the necessary permissions required by the ad library; the developer must make sure the ad-supported app requests these permissions by making the necessary changes to its manifest file. Similarly, in order to be paid for the ads served by the app, the developer must make sure the ad library is furnished with their developer identifier.

Angry Birds’ manifest file (included as Figure 5.1) provides a representative example of this arrangement. This particular version of Angry Birds contains Google’s popular AdMob ad library, which pulls some of its control data from the manifest of its host app. Such data includes the crucial publisher identifiers, which are stored as the “ADMOB_PUBLISHER_ID” and “ADMOB_INTERSTITIAL_PUBLISHER_ID” meta-data values. Also in the manifest file, AdMob listens for package installation events by registering the com.admob.android.ads.analytics-
InstallReceiver component, and defines its own Activity (screen) with com.admob.android.-ads.AdMobActivity to display full-screen ads.

In general, ad libraries can be classified into three ad-oriented categories: mobile web libraries, rich media libraries, and ad mediators. Mobile web libraries are front-ends to web-based ad networks. Content is requested, delivered and displayed using standard web technologies, with very little interaction with the device’s APIs. These libraries typically display only banner or text ads. In our study, we found over half of existing in-app ad libraries are of this type. Rich media libraries have a similar mission, but behave more like powerful platforms. Specifically, they provide feature-rich APIs for both app developers and advertisers. While they can display the simpler ad types, they can also support more advanced kinds such as active content (i.e., JavaScript), video, interstitial ads and the like. Although there are fewer ad libraries as rich media libraries than mobile web libraries, many of the most popular ones, including AdMob, are actually rich media ones. The third category, ad mediators, is different from the previous two by exposing a standard interface through which an app developer can interact with other ad libraries of the other two types. Since ad libraries often request similar information from the app developer in very different ways, these mediator libraries exist to make bundling multiple ad libraries in an app easier.

Our experience indicates that all three kinds of ad libraries tend to share some common characteristics. For example, they have user-interface code (to present their ads) and network code (to request ads from the ad network’s servers). They are also designed to be tightly bundled with host apps. In this way, it becomes more difficult to disable the ad functionality or defraud the ad network. To the same end, some ad libraries heavily obfuscate their internal workings in an effort to discourage reverse engineering. AdMob again provides a representative example. Inside the AdMob ad library, only the classes, methods and fields described in the AdMob documentation have meaningful names; everything else has had its name changed to a letter of the alphabet. Moreover, all debugging information is stripped from all the classes in the package.

<table>
<thead>
<tr>
<th>Protection level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>normal</td>
<td>Low-risk permissions granted to any package requesting them</td>
</tr>
<tr>
<td>dangerous</td>
<td>High-risk permissions that require user confirmation to grant</td>
</tr>
<tr>
<td>signature</td>
<td>Only packages with the same author can request the permission</td>
</tr>
<tr>
<td>signatureOrSystem</td>
<td>Both packages with the same author and packages installed in the system image can request the permission</td>
</tr>
</tbody>
</table>
At runtime, the embedded ad libraries execute together with the host app inside the same runtime environment – a Dalvik [12] virtual machine (VM), which is eventually instantiated as a user-level process in Linux. Different apps run in different Dalvik VMs, isolated from each other. The Dalvik VM is derived from Java but has been significantly revised (with its own machine opcodes and semantics) to meet the resource constraints of mobile phones. When an app is installed in Android, it is assigned its own unique user identifier (UID) – as Android relies on the Linux process boundary and this app-specific UID assignment strategy to achieve isolation or prevent a misbehaving or malicious app from disrupting other apps or accessing other apps’ files. Unfortunately, this strategy does not separate host apps from the in-app ad libraries they contain, as those libraries inhabit the same Dalvik VM and execute with the same UID. In our example, AdMob could readily send the user’s Angry Birds scores to Google.

The situation is further complicated by the fact that Android apps are structured differently than programs on most platforms, in that they can contain multiple entry points. These entry points are invoked by the framework in response to inter-process communication (IPC) events; even “running” an app is treated in this way. Technically, each app is composed of one or more different components, each of which can be independently invoked. There are four types of components: activities, services, broadcast receivers and content providers. An activity represents part of the visible user interface of an app. A service, much like a Unix daemon, runs in the background for an indefinite period of time, servicing requests. A broadcast receiver receives and reacts to broadcast announcements, while content providers make data available to other apps. Each Android app is deployed in the form of a compressed package (apk). These apk files contain a manifest file (AndroidManifest.xml) that describes various standard properties about the app, such as its name, the entry points (or interfaces) it exposes to the rest of the system, and the permissions it needs to perform privileged actions. The Angry Birds manifest (Figure 5.1) describes two entry points defined by AdMob instead of Angry Birds: an activity (com.admob.android.ads.AdMobActivity) and a broadcast receiver (com.admob.android.ads.analytics.InstallReceiver). The activity is designed to be invoked by the code in the app, but the broadcast receiver is interested in com.android.vending.INSTALL_RE duces events sent out by the Google Play app. Accordingly, it’s possible to invoke the ad library’s code directly before any of the host app’s code is run.

To better protect personal information and manage system resources, Android defines a permission-based security model [5]. In this model, the principals that have these permissions are apps, not users or libraries. The Android framework contains a pre-defined set of permissions and also allows developers to define additional permissions as they see fit. Each permission has a protection level [3], which determines how “dangerous” the permission is and what other apps may request it. Table 5.1 summarizes the defined protection levels in Android. The signature and signatureOrSystem permission protection levels are reserved to define capabilities that are
not meant to be used by apps written by other authors or by apps that are part of the system image. Permissions are checked either through annotating entry points defined in the manifest file or programmatically by the Android framework. Since ad libraries are not principals, they inherit the permissions of the apps they are embedded in. As a result, many ad libraries opportunistically check for and use permissions. Some may allow the host app’s author to control their behavior somewhat while most ad libraries simply use what permissions granted to the host apps.

5.3 System Design

The goal of this work is to assess possible privacy and security risks posed by the embedded in-app ad libraries and additionally quantify these risks by measuring their prevalence on Android. Note that the Android’s permissions-based security model provides a convenient way to measure the risk inherent in Android APIs, as their documentation typically mentions whether a permission check is required to successfully make the call. However, as mentioned previously, ad libraries are not annotated in any way by the Android framework. Also, the context surrounding each potentially-dangerous Android API call is very important in matters of privacy. For instance, if the user’s phone number is retrieved but never sent to the Internet, no privacy violation has occurred. In this work, we opt to crawl and collect available apps from Google Play. After that, we systematically identify representative ad libraries from these apps and then develop a system to thoroughly identify possible risks. Figure 5.2 summarizes the methodology in our study.

5.3.1 Sampling Google Play

We crawled Google Play, then known as the Android Market, for apps over three months (March through May, 2011) and chose the first 100,000 downloaded apps as the dataset for our study. With them, we built a database that extracts the features needed to perform our later analysis,
i.e., the permissions requested by each app (as defined in its manifest file) as well as the Java class tree hierarchy contained in the app’s code.

After that, among the 100,000 apps, we select apps that have the `android.permission.-INTERNET` permission, which is required for communication with the ad network’s servers, and organize them into a candidate set. From the candidate set, we randomly select an app and disassemble it. The disassembled bytecode is examined for new ad libraries. Especially, in the search process for new ad libraries, we maintain an ad set, which is initialized to be empty. For each new ad library we identified, we add it to this set. Further, we extract its unique class tree and use it as the pattern to detect the list of host apps that contain this particular ad library. Specifically, we remove those host apps from the candidate set. We repeat the selection process until 100 distinct ad libraries have been selected. By searching the class trees stored by the database for each ad library’s package name, we can then determine how many apps within our sample of 100,000 contain the given ad library. Sorting and graphing these figures of the top 20 ad libraries produces the graph in Figure 5.3. (The list of 100 ad libraries is detailed in Tables 5.2 and 5.3 – Section 5.4.) In total, the 100 ad libraries in our study are present in 52.1% of the collected 100,000 apps.

Among these 100 representative ad libraries, Google’s own AdMob, AdSense, and Analytics networks are listed in the top five. We also note that several other networks – Flurry, MillenialMedia, Mobclix, and AdWhirl – appear in a comparatively large number of apps. Given
the maturity of these ad networks behind these leading libraries, we expect that the libraries themselves offer standard functionality and do not engage in activities frowned upon by the industry as a whole. On the other hand, any potential privacy risks posed by such commonly-deployed libraries would impact many users. Among the remaining libraries, there allegorically appear to be a large number of small ad networks that offer in-app ad libraries on Android. The large number of such libraries, coupled with the relatively small proportion of apps they appear in, make holding their behavior to account more difficult for watchdog organizations inside and outside the ad industry. Analyzing these libraries is therefore important in order to gain perspective on the range of behaviors ad networks will engage in.

5.3.2 Analyzing Ad Libraries

After identifying the 100 representative ad libraries, we next seek to determine whether a given ad library contains any risks to security or privacy. To do that, we start by considering the permission protection levels [3] defined by the standard Android framework. Note that various standard APIs exposed by the framework require certain permissions to access, which have been annotated by a protection level. Any APIs that require a permission with an elevated protection level (i.e., above “normal”) can be considered a risk to security or privacy.

Unfortunately, the relationship between APIs and permissions can be difficult to determine. The Android documentation does not feature an exhaustive list of these relationships, and some permissions are only conditionally checked. For example, Android defines two related permissions that allow access to the user’s location data: `android.permission.ACCESS_COARSE_LOCATION` and `android.permission.ACCESS_FINE_LOCATION`. Both permissions are checked by the methods of the `android.location.LocationManager` class; however, these methods determine which permission to check by the arguments they are given. For example, calling `LocationManager.getLastKnownLocation('gps')` requires the `android.permission.ACCESS_FINE_LOCATION` permission; the same call with the argument of ‘‘wifi’’ would instead require the `android.permission.ACCESS_COARSE_LOCATION` permission.

To address these challenges, we apply and extend Felt et al. [91] to derive a list of API calls that are of interest for our analysis. In particular, we take a similar approach by analyzing the Android documentation, source code and disassembled bytecode to conservatively annotate the standard APIs with the permissions that they require. However, unique to this study, our extensions also include a new set of Android API calls, which do not require any permission (Section 5.4). In particular, most of them are related to `ClassLoader` and reflection mechanisms. The `ClassLoader` part is responsible for dynamically loading code at runtime. To elaborate, in Dalvik, class references are resolved at run-time. Usually, due to the presence of a verifier looking for undefined references, it is safe to consider a Dalvik app as containing only well-
defined static code. When combined with reflection API, it becomes possible to refer to classes using data at run-time, thus invoking the `ClassLoader` functionality after the verifier has run. Since the `ClassLoader` is itself just a class, its methods can be overridden to allow developers to pass raw bytecode to the Dalvik VM at run-time. In this fashion, it is possible to download and run arbitrary dynamic code, rendering any static analysis of an app incomplete. Fortunately, the interfaces to the underlying Dalvik VM are well-defined. We treat these interfaces as just another kind of APIs, which not only implicitly marks dynamic code loading as a suspicious behavior, but unifies our analysis framework. In total, our current system considers 76 distinct permissions (34 dangerous, 26 signature, 11 signatureOrSystem, and 5 normal – Section 5.4).

5.3.3 Identifying Possible Risks

After identifying the set of APIs of interest, we then perform a reachability analysis for each ad library. We are interested in two dimensions of potentially dangerous behaviors, which means we must deal with up to four potential reachability conditions. The first dimension involves the precipitating event for the dangerous behavior; that behavior could come from one of Android’s many entry points, or could be in response to a received network packet. Finding a path from either of those start points to an API could signal a dangerous situation, but may not necessarily; this is where the second dimension comes into play. Some API calls are dangerous in themselves (such as those that can cost money) while others merely expose personal data that can then be leaked to an external party. In the first case, finding a path from an initiating entry point or network connection is sufficient, but in the second we must further find a dataflow path from the dangerous call to an external sink (e.g., network APIs).

In mechanical terms, our method is as follows: each ad library sample’s bytecode is first scanned for the dangerous API calls we previously annotated. For each found API call, we trace backwards through the library source looking for potential entry points and any mitigating circumstances; for example, if such a dangerous API call only occurs if a flag representing the user’s consent is set, we note this behavior. Some API calls may not be reachable under any circumstances and therefore may be safely ignored, but all others are recorded if they match these conditions. For those calls that leak information, we then additionally trace forwards through the bytecode looking for a network sink. If one is found, the candidate path from the API call to the network is also recorded\(^1\). In algorithmic terms, we produce a control-flow graph showing all the possible paths of execution through the library, then determine which of those paths are indeed feasible.

\(^1\)Note that we do not ignore calls that do not meet this additional criterion due to the complexities inherent in dataflow analysis. It is possible to introduce dataflow discontinuities using threading, caching, and other behaviors; we elect to involve some additional manual effort in order to ensure the accuracy of our study.
In our prototype, we leverage the existing **baksmani** Dalvik disassembler [7] to automate some of this process. As part of the greater **smali** package, this allows us access to a convenient intermediate representation and a limited set of intra-procedural static analysis tools. Using it as the base, we add code to derive the control-flow graph which we will traverse to find the set of feasible paths through the app (and thus the ad library).

Due to a key difference between Android apps and traditional Java programs, traversing the derived control-flow graph poses additional challenges under Android. Specifically, Java programs, like those written in many other languages, start execution at a **main** method. Android apps have no such method, instead containing a number of entry points based on the components they contain (certain methods in e.g., **Service** and **Receiver** objects). In addition to these, the library itself usually exposes some methods to the host app for initialization purposes. The entry points specified by the framework are automatically identified on the basis of the class they belong to, while the library’s initialization methods are fed into the system through annotation. We then run the subsequent steps in our analysis over each entry point in turn, finally merging the results.

Our experience indicates that due to the influence of native code and the core classes in Android framework (e.g., the use of threads – a common technique in Android for improved user responsiveness), we observe discontinuities in the generated call graph. To resolve these discontinuities, we elect to load an additional set of class files alongside the library. These files stand-in for core classes and contain simple expressions designed to capture the semantics of each API call. Additionally, these files include the dangerous API calls the system is supposed to identify; each dangerous call contains a sentinel instruction that alerts our analysis code to its nature for the next stage of analysis.

Given this control-flow graph, our algorithm next attempts to find reachable paths from an entry point to a dangerous API call. To do this, we perform the traditional information-flow analysis, where constraints are placed on the variables and checked against by branch instructions. In the resulting feasible control-flow graph, we verify whether each dangerous API call is in a feasible code region. If a call is, execution is traced backwards and the necessary constraints remembered to form an execution path, which is then reported. The paths reported by our system are then verified. The reentrant, multi-threaded nature of Android apps makes points-to analysis difficult, which in turn frustrates efforts to accurately identify only feasible paths through the library. Certain language features are not fully supported yet in our current prototype. For example, the Java Reflection APIs (i.e., the **java.lang.reflect.** package) allow code to be invoked by name, and without perfect dataflow analysis tools this causes an irreconcilable discontinuity in the generated control-flow graph. To accommodate such situations, we take a conservative approach wherever possible, preserving accuracy but necessitating
additional manual effort on some occasions. In particular, we report the use of reflection APIs in an ad library to highlight their presence for further investigation (Section 5.4).

5.4 Prototyping and Evaluation

We base our static analysis tool on the open-source baksmali Dalvik disassembler (version 1.2.6). Implementing the design laid out in the previous section required 2809 new lines of code and four hooks in the original baksmali project. As stated in the design section, our system also required each API of interest to be annotated so that it could be analyzed by the system. Accordingly, we annotated APIs associated with 76 standard Android permissions. As our static analysis approach is rather standard, in the following, we mainly focus on the peculiarities of the Android platform and the new extensions we added for risk analysis.

Specifically, besides reporting potentially-feasible paths, our prototype has been extended to report on five other code patterns of interest: the use of reflection, dynamic code loading, permission probing, JavaScript linkages and reading the list of installed packages. As the presence of one or more of these patterns can color our other findings for a given library, we opt to have our tool automatically report them alongside its feasible-path output.

The first such pattern, the use of reflection, concerns the use of the java.lang.reflect package. As mentioned earlier, this portion of the Java specification allows programmatic invocation of methods and access to fields, which complicates our static analysis. Without it, the static analysis of Dalvik bytecode is reliable and unambiguous. In theory, reflection essentially makes resolving an app’s call graph into a dataflow problem. In practice, often reflection appears to involve constant strings, thus introducing no new ambiguity. However, this is not always the case in our collected ad samples. Therefore, our system makes what assumptions it can while flagging the situation for further review.

In a similar vein, Android apps usually are amenable to static analysis techniques because they are designed to be loaded as a whole and statically verified by the framework itself. However, another esoteric Java language feature was carried over into Dalvik: the ClassLoader class. This class is used by the framework to find code resources on demand. Usually, the static verification stage causes practically the entire app to be loaded at once, as the verifier attempts to resolve all the references in the bytecode. However, using reflection, it is possible to cause a class to be loaded that is not directly referenced by any existing code. As the ClassLoader class can be extended by developers, custom versions of this class can be written to load code from non-standard resources. Each such ClassLoader inherits from a parent version of the class, on up to the baseline “system” instance of the class. Since Dalvik, unlike Java, does not permit the “system” ClassLoader to be changed by the developer, dynamic code loading is very explicit: the generic reflection API cannot be used to implicitly reference a class for the
first time, so instead the custom ClassLoader must be explicitly queried. Our prototype flags this behavior and raises a serious warning, as its presence negates all existing static analysis efforts and signals suspicious dynamic code loading behavior.

A more common pattern our prototype elects to handle specially is what we call “permission probing.” In this pattern, an ad library contains some API which requires permission to successfully call. Instead of mandating that the developer of the host app requests this permission, the ad library can instead opportunistically attempt to use the API, either by checking that it has the necessary permission beforehand, or by handling the SecurityException that is thrown by most APIs when they are called with insufficient permission. These methods of checking permissions are well-defined under Android, and so we can inspect the control-flow graph to detect branches that detour around dangerous API calls.

Similarly, it seems to be common practice for “rich media” ad libraries to offer JavaScript bindings to expose additional functionality to JavaScript ads. We elect to include this practice in our findings for two reasons. Not only is this behavior indicative of “rich media” libraries, it also raises interesting privacy concerns, which we will cover in greater depth in Section 5.4.2.

Lastly, we temper our results by showing one instance of an invasive API that, for whatever reason, requires no special permission to access. Some ad libraries we studied collect the list of all apps installed on the device. This information is every bit as personal as the user’s browser history, in that it reveals some information about their interests. We include this behavior to demonstrate the guile advertisers have and the incompleteness of the permission-based system.

In the rest of this section, we present our findings from the analysis of 100 representative ad libraries. We first summarize our results in Section 5.4.1 and then present categorized findings about dangerous behaviors in these ad libraries in Section 5.4.2. Finally, we evaluate the performance of our prototype in Section 5.4.3.

### 5.4.1 Overall Results

Before tabulating our findings, we stress that our results are for ad libraries rather than apps. Some apps in our sample of 100,000 will contain more than one ad library, while others contain none at all. For the 100 representative ad libraries in our study, we found that they are embedded in 52,067 different host apps in our dataset. As one host app may contain more than one ad library, we show a breakdown of how many libraries each of these apps contain. The result is shown as Figure 5.4.

From the figure, it seems more than one third of apps (or more precisely 34,130) contain one ad library and a small fraction of apps (around 3%) include at least five ad libraries for monetization. One particular host app, i.e., com.Dimension4.USFlag, embedded no fewer than
Table 5.2: The overall results from the top 50 representative ad libraries

<table>
<thead>
<tr>
<th>Ad Library</th>
<th>Overall Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>admob</td>
<td>27235</td>
</tr>
<tr>
<td>google/ads</td>
<td>16323</td>
</tr>
<tr>
<td>flurry</td>
<td>5152</td>
</tr>
<tr>
<td>google/.../analytics</td>
<td>4551</td>
</tr>
<tr>
<td>millennialmedia</td>
<td>4228</td>
</tr>
<tr>
<td>mobelix</td>
<td>4100</td>
</tr>
<tr>
<td>admiral</td>
<td>3915</td>
</tr>
<tr>
<td>qwapi</td>
<td>1745</td>
</tr>
<tr>
<td>youmi</td>
<td>1699</td>
</tr>
<tr>
<td>moboxy</td>
<td>1524</td>
</tr>
<tr>
<td>zetada</td>
<td>1514</td>
</tr>
<tr>
<td>cashly</td>
<td>1249</td>
</tr>
<tr>
<td>inomi</td>
<td>1229</td>
</tr>
<tr>
<td>wooobo</td>
<td>1183</td>
</tr>
<tr>
<td>admarvel</td>
<td>1101</td>
</tr>
<tr>
<td>smaato</td>
<td>1077</td>
</tr>
<tr>
<td>mobclick</td>
<td>1058</td>
</tr>
<tr>
<td>nobot</td>
<td>995</td>
</tr>
<tr>
<td>airpush</td>
<td>945</td>
</tr>
<tr>
<td>madtm</td>
<td>883</td>
</tr>
<tr>
<td>vdocpa</td>
<td>872</td>
</tr>
<tr>
<td>wiyun</td>
<td>777</td>
</tr>
<tr>
<td>android</td>
<td>651</td>
</tr>
<tr>
<td>madhouse</td>
<td>603</td>
</tr>
<tr>
<td>pontiflex</td>
<td>522</td>
</tr>
<tr>
<td>innerActive</td>
<td>497</td>
</tr>
<tr>
<td>adserver</td>
<td>492</td>
</tr>
<tr>
<td>casee</td>
<td>479</td>
</tr>
<tr>
<td>greystripe</td>
<td>440</td>
</tr>
<tr>
<td>omniture</td>
<td>433</td>
</tr>
<tr>
<td>guohead</td>
<td>400</td>
</tr>
<tr>
<td>daum</td>
<td>309</td>
</tr>
<tr>
<td>domob</td>
<td>374</td>
</tr>
<tr>
<td>tapjoy</td>
<td>368</td>
</tr>
<tr>
<td>adlantis</td>
<td>341</td>
</tr>
<tr>
<td>adagogo</td>
<td>339</td>
</tr>
<tr>
<td>adchina</td>
<td>327</td>
</tr>
<tr>
<td>jumptap</td>
<td>278</td>
</tr>
<tr>
<td>medialets</td>
<td>274</td>
</tr>
<tr>
<td>nowistech</td>
<td>272</td>
</tr>
<tr>
<td>waps</td>
<td>239</td>
</tr>
<tr>
<td>vpon</td>
<td>189</td>
</tr>
<tr>
<td>energysource</td>
<td>160</td>
</tr>
<tr>
<td>iconowsys</td>
<td>131</td>
</tr>
<tr>
<td>adwo</td>
<td>131</td>
</tr>
<tr>
<td>sktelecom</td>
<td>125</td>
</tr>
<tr>
<td>upbosad</td>
<td>112</td>
</tr>
<tr>
<td>smartadserver</td>
<td>102</td>
</tr>
<tr>
<td>airad</td>
<td>89</td>
</tr>
<tr>
<td>hamiantx</td>
<td>85</td>
</tr>
</tbody>
</table>

Total: 92297
Table 5.3: The overall results from the remaining 50 ad libraries

<table>
<thead>
<tr>
<th>Ad Library</th>
<th>Included in Apps</th>
<th>Permissions</th>
<th>Users</th>
<th>Installation Data</th>
<th>Location</th>
<th>Call Logs</th>
<th>Contacts</th>
<th>Calendar</th>
<th>SMS</th>
<th>Change Calendar</th>
<th>Contacts</th>
<th>ClassLoader</th>
</tr>
</thead>
<tbody>
<tr>
<td>mopub</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>adfonic</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vddroid</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>transpera</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mobgold</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mobus</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hutuchong</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mads</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>everbadge</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>zetacube</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>livepoint</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l/adlib_android</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>trickersticks</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ccmedia</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>soscoo</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mojiva</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>admogo</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nexage</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rhythmnewmedia</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mediba</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>madvertise</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>qriously</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>zumobi</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>netmite</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imapp</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>yicha</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>oneriot</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>aduru</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>admoda</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imobile</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>moblico</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>linkshare</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ignitevision</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fractalist</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>plankton</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>celltads</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>appmedia</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>glam</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>adttu</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>adfox</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>netad</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>freewheel</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>adinside</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>taobao</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>amaze</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>admobi</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wqmobile</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ubermind</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>innoace</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AdyxSilk</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>1239</td>
<td>24</td>
<td>14</td>
<td>9</td>
<td>51</td>
<td>22</td>
<td>70</td>
<td>1</td>
<td>0</td>
<td>41</td>
<td>125</td>
<td>91</td>
</tr>
</tbody>
</table>
20 ad libraries! However, it is unclear whether the inclusion of more ad libraries necessarily brings more profit to an app’s developers.

Our system scans each representative ad library for the use of 76 dangerous APIs. The overall results for the top 50 ad libraries are shown in Table 5.2, while the results for the remaining 50 are shown in Table 5.3. In practice many of Android’s dangerous APIs were not used by any ad library, so we opt to omit such APIs for brevity in our results. Specifically, the two tables contain the 14 dangerous APIs we see used by at least one ad library. In the tables, we also include data on six structural properties of interest, such as the use of obfuscation, conditional API use via permission probing, and dynamic code loading through the ClassLoader language feature. Overall, our system reports 318 total API uses and structural patterns. Upon further verification, 19 of them ask permission from the user and our system properly recognizes 15 of these cases, which happen to be all related to text message (SMS) API calls.

Despite all of the reported APIs being marked as “dangerous,” our results show that some APIs are commonly used by ad libraries. These include the location APIs and a single “Read Phone Information” API call, both of which are used by at least half of the ad libraries we analyzed. The ad libraries use these APIs for targeting information: the location APIs can be used to serve ad content relevant to the geographic area of the user, while the commonly-used phone information call returns a unique identifier (the phone’s IMEI number) that is useful for tracking what content has been served to a particular user. These uses seem plausible, in the context of an ad library; however, we did identify two ad libraries (Mobclix and adserver) that expose this information directly to advertisers, which is harder to justify.
The remaining dangerous APIs either provide some feature, or allow access to more intrusive data maintained by the device. The feature-based APIs appear to be mostly harmless. For example, a number of ad libraries allow ads to place phone calls, send text messages or add an event to the calendar. In all of these cases, these functions are performed only after the user triggers them (i.e., by clicking on an ad) and confirms their intentions.

More insidious, however, are requests for information that is not directly useful for ad targeting. Our analysis uncovered a few instances where an ad library accessed information that is only useful when correlated with other facts known about the user. This correlative information is a direct threat to the user’s privacy, because it can be used to uncover the user’s true identity. For example, it is hard to make a case that the user’s call history has any bearing on what ads they will be interested in, yet we discovered one ad library (sosceo) transmitting some of that information to the Internet (to be detailed in Section 5.4.2). In a similar vein, a large number of ad libraries used an API call to retrieve the user’s phone number, and another ad library (Mobus) peculiarly reads through the user’s SMS messages to determine which text-messaging service center they use. Finally, we identified one particularly worrying use of an otherwise innocent API, where some ad libraries (such as waps) upload a list of all the installed apps on the phone.

Looking beyond privacy concerns, we identified five ad libraries which make use of the ClassLoader feature to dynamically load code at runtime. These ad libraries are effectively impossible to statically analyze as a result; at a whim, their code can be changed. A malicious or compromised ad network could command its ad libraries to download a botnet payload or root exploit, for example. Our later investigation indeed captures one suspicious payload, which essentially turns the host app into a remotely-controllable bot (Section 5.4.2).

Moreover, the other structural properties of ad libraries are worth mentioning. Over half of the ad libraries we studied employed obfuscation techniques, presumably to discourage reverse engineering. While not altering the function of the library, these transformations strip human-
readable names from methods and classes while optionally muddling the control flow by adding pointless redundancy or by reordering instructions. As an example, we list in Figure 5.5 the classes contained in one particular ad library, i.e., AirAD. Only a few such classes have names that carry any meaning; all the rest are strings of “l” (lowercase L) and ‘t’ characters. The ad libraries that are noted as using obfuscation in Table 5.2 all used some scheme to obfuscate their internal classes, and typically also obfuscate the names of fields, methods and the like in a similar fashion. Other obfuscators are known to exist, but all serve the same purpose; for example, the default obfuscator names classes after alphabetical characters, while another uses nonsense dictionary words like “Watermelon” and “Railroad.” Applying these techniques to a reasonably large ad library hides the intent behind much of what the library does, while not truly protecting the ad network’s trade secrets – as the library can still be unambiguously analyzed, and the network’s core functionality resides on its servers regardless, safe from competitors’ eyes.

In another common pattern, many ad libraries probe the permissions available to them before attempting to use permission-guarded APIs. Normally, if an Android app calls an API it does not have permission to access, a `SecurityException` is thrown. If this exception is not caught, the app will crash. In order to prevent this from happening, ad libraries either check their permissions up front or silently catch the thrown exception. It turns out more than half of studied ad libraries (marked in Tables 5.2 and 5.3) engage in this sort of behavior. Some of them do log their failed attempts to access these APIs, chastising the host app’s developer for not properly requesting the necessary permissions. However, most attempt to do what they can with as many permissions as they can access, again silently. A few libraries, such as AdMob, do permit the host app developer to selectively deny the library permission to use a certain API. This is unfortunately far from the norm, and only Mobclix allows the user to disallow access to sensitive APIs – on a case-by-case basis, and with some troubling ramifications, as elaborated in Section 5.4.2.

Lastly, some ad libraries use the Java reflection language feature, which essentially allows programmatic access to methods and fields by their name. Normally, when Dalvik bytecode is loaded, there is a static verification step that ensures all referenced code elements are valid. Reflection sidesteps this mechanism, which allows for the use of dynamic code (discussed at more length in Section 5.4.2), but can also be used to access any code that is not guaranteed to resolve correctly on all devices. In this way, it is possible for ad libraries to access “experimental” APIs or vendor-specific APIs. Given the lower maturity of such APIs, their use by ad libraries is suspect.
5.4.2 Categorized Findings

To provide greater detail about problematic behaviors we came across in our analysis, we organize them into three categories.

Invasively Collecting Personal Information

The first category involves the questionable collection of personal information. Specifically, some ad libraries brazenly request information not directly useful in fulfilling their purpose. Our results show that the larger ad networks typically do not engage in such questionable activities, but smaller ad networks might. Unfortunately, there is no way for the user of an app to know which ad networks it contains.

A representative example of this behavior can be found in the sosceo ad library, one of the least popular libraries studied. Like most ad libraries, sosceo is instantiated by its hosting app making a UI element designed to display an ad, in this case a `com.sosceo.android.ads.-AdView` object. When this object is created, a fairly lengthy set of obfuscated method calls occurs. These method calls ultimately query the device’s contact information database for the most recent phone call. This information is duly stored in a field of a data object used by the `AdView` object; when the `AdView` object requests an ad from the backing ad network, the information is included as an URL query string under the “dp” key to the ad server.

Other ad libraries engage in similarly strange behaviors; Mobus, for example, reads the SMS (text-message) database looking for administrative information about the user’s Short Message Service Center (SMSC). This SMSC is the back-end service provider responsible for routing text messages to and from the user. For some unknown purpose, Mobus transmits this information to its servers.

Similarly, Pontiflex takes an interest in what account credentials the user has on the device. This information is not a direct security risk as the ad library does not have access to the credentials themselves, but it does query the list of accounts the user’s phone manages. Somewhat suspiciously, the dangerous API calls in this case are performed via the reflection API, which is a language feature that allows methods to be invoked by means of data strings. It is possible that reflection is being used, in this case, to throw off static analysis of the library.

Permissively Disclosing Data to Running Ads

The second category involves the direct exposure of personal information to running ads. One of the most popular ad libraries, Mobclix, appears at first glance to function like most other ad libraries. To display an ad, Mobclix creates an `android.webkit.WebView`, which is essentially a miniature web browser. The ad is then rendered by this web browser for display, allowing the advertiser to design their ads using standard web technologies.
However, unlike its principal competitors, Mobclix attempts to gain advantages by offering its advertisers access to certain smartphone features. Since these features do not have standard hooks in HTML or JavaScript, the Mobclix ad library has a class (`com.mobclix.android-sdk.MobclixJavascriptInterface`) that binds certain Android APIs to JavaScript functions that are then exposed to ads rendered within the WebView. Each API call is wrapped in a method that simply and succinctly exposes it to JavaScript. By doing so, Mobclix exposes a great variety of API calls and allows running ads to most of the sensors and data on the phone. Note that most of these accesses include appropriate user confirmation dialogs. For example, while an ad can call `contactsAddContact(...)` to add a contact to the user's address book, nothing will happen unless the user gives consent via a dialog box.

Unfortunately, not all functions are safely wrapped in this way. For instance, the `gpsStart(...)` function allows a JavaScript ad to register a callback function. This function will be called immediately, and again whenever the user’s moves more than a defined distance from the last reported position. The user is never asked for their consent, nor are they notified in any way that this feature of their phone is being used by an ad. This particular example sufficiently raises interesting privacy issues. It is reasonable to expect that the Mobclix ad library itself should have access to location information; such information is commonly used to target ads to a certain geographical area. However, this code is not actually using that information for Mobclix’s ad-targeting purposes. Instead, the information is being given to a third party advertiser. Indeed, given access to this functionality, the ad itself can be thought of as dynamically-loaded code of unknown provenance (Section 5.4.2).

### Unsafely Fetching and Loading Dynamic Code

The third category involves unsafe fetching and loading of dynamic code (possibly from the Internet), which poses an even greater potential threat for two reasons. One is that this dynamically loaded code cannot be reliably analyzed, effectively bypassing existing static analysis efforts. The other is the fact that the downloaded code can be easily changed at any time, seriously undermining the capability of predicting or confining its behavior.

In the 100 representative ad libraries, five of them have this unsafe practice. One particular one will be downloading suspicious payloads, which allows the host app to be remotely controlled. Specifically, the portion of this ad library that is embedded in the host app is very small: a single service, `com.plankton.device.android.service.AndroidMDKService`. This service contacts a remote server with the list of permissions granted to the host app and the phone’s hardware identifier (IMEI); in return, the remote server provides it with the URL to download a `.jar` file (see Figure 5.6). This `.jar` file contains the vast majority of Plankton’s code, which is then dynamically loaded using a `dalvik.system.DexClassLoader` object – Dalvik's base
Figure 5.6: Handshake communication between Plankton and its command-and-control server implementation of the ClassLoader Java language feature. The downloaded .jar will listen to remote commands and turn the host app into a bot. Based on this discovery, we have reported the seven affected host apps to Google, which promptly removed them from the market on the same day.

This behavior is interesting because it highlights the dynamically-linked nature of Dalvik. Android apps are distributed as bytecode, which makes app analysis easier due to the clearly-defined semantics of the format. Furthermore, upon loading a class for the first time, a Java-style bytecode verifier makes certain that all references within the class resolve. This verification step seems to preclude adding arbitrary code at runtime. However, via the java.lang.reflect package, Java (and hence Dalvik) can load classes by name at runtime. Coupling this language feature with the ability to control where Dalvik looks for definitions for such classes – that is, the DexClassLoader class – allows apps to load arbitrary code not contained in the app’s package file. In this case, the downloaded .jar file has a predefined entry point, com.plankton.device.android.AndroidMDKProvider.init(...). DexClassLoader looks for it by name and then invokes the control logic. Within the newly-downloaded code, the bytecode verifier works as usual, since it now uses the modified DexClassLoader to resolve references to unfamiliar classes.

Another four ad libraries make use of this feature, likely as a version-control and content-delivery mechanism. Opening the full expressive power of Dalvik – replete with all the permis-
Table 5.4: Processing time of analyzed ad libraries

<table>
<thead>
<tr>
<th>Library</th>
<th>Processing Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdMob</td>
<td>16.17s</td>
</tr>
<tr>
<td>AdWhirl</td>
<td>17.25s</td>
</tr>
<tr>
<td>Appmedia</td>
<td>14.58s</td>
</tr>
<tr>
<td>Quattro</td>
<td>14.40s</td>
</tr>
<tr>
<td>UplusAd</td>
<td>15.91s</td>
</tr>
</tbody>
</table>

sions granted by the app – to nebulously downloaded dynamic code has unfortunate privacy implications. Again, given that the code retrieved from the Internet will naturally change, it is impossible to verify that the ad library is only engaging in the behaviors embodied in the library.

5.4.3 Performance Measurement

Next, we report the performance overhead of our prototype. In our test, we picked up five ad libraries and run our system to analyze each of them ten times. Each analysis run scans the given ad library for all (80) APIs our prototype handles. In each run, we record the processing time and report the average. Our test machine is an AMD Athlon 64 X2 5200+ machine with 2GB of memory and a Hitachi HDP72502 7200 rpm hard drive. We summarize the results in Table 5.4. The test-case libraries were selected to provide a mix of ad library types and complexities. Each library took, on average, \( \sim 15.66 \) seconds to process. Given our tool is designed to be used in an offline, semi-automated capacity, we believe this performance to be acceptable for our purposes.

5.5 Discussion

Our study has so far uncovered a number of serious privacy and security risks from existing in-app ad libraries on the popular Android platform.\(^2\) Given this, it is important to examine possible root causes and explore future defenses.

First, due to the fact that ad libraries are incorporated into the host apps that use them, they in essence form an symbiotic relationship. Based on such relationship, an ad library can effectively leverage it and naturally inherit all permissions a user may grant to the host app, thus undermining the app-based privacy and security safeguards. Accordingly, we believe that the exposed risks are fundamentally rooted in the granularity problem in the essential Android’s permissions model. Under this model, the smallest entity that can be granted a permission is an app. Even though ad libraries come from a different developer and have different intentions

\(^2\)While we only studied one particular platform, due to the similar nature of integrating in-app ads into smartphone apps, we expect similar privacy and security risks will also exist on other platforms.
than their hosting apps, they are afforded the same permissions. As we have seen, advertisers themselves are sometimes allowed to execute code within an app, adding yet another untrusted set of principals to the list of parties covered by a single permissions policy. Though an app’s requests for access to private information can stem from the app’s code, the ad library’s code, or both, but the user or rather the Android platform cannot determine at a glance which parties will use the information.

Second, the current situation could also be a product of one central tension: the same solutions that would allow ad libraries to be sandboxed could also be used to disable them, or alternatively, defraud them. Even if Google had provided a separate Advertiser template in the Android framework (i.e., alongside the Services, Receivers, ContentProviders and Activities that exist today), there would be no incentive for ad networks to use it. It is safer to tightly couple ad libraries with their host apps, to keep them from being easily circumvented. Possibly for the same reason, some ad networks take the approach of the worrisome dynamic code loading behavior we observed. In particular, since ad libraries are not their own entity in the framework, they can only be updated alongside their host app. The ad network cannot control the release schedule of all the apps its ad library is bundled with. As a result, any code updates need to be pushed out alongside channels. The dynamic code loading apparently becomes the choice at the cost of raising privacy and security concerns to mobile users.

Third, we may also consider ways to design ad libraries that satisfy the needs of advertisers, ad networks and users alike [102, 103, 168]. As in traditional web-based ad libraries, these systems display targeted advertising and report the network impressions, click-throughs, etc. to bill the advertiser. However, they aim to do these things irrefutably yet anonymously. The ultimate aim is to only provide the ad network with the metrics needed for billing, while allowing the user to retain complete and direct ownership of personally-identifiable information. Unfortunately, each approach proposed so far has required either additional overhead (extra data transfers, extra storage on the device, etc.), an organizational shift (third-party ad “dealers,” the direct involvement of wireless providers, etc.), or both. As some ad libraries may not brand the ads that they serve, the user is usually ignorant of the ad networks used by an app. Therefore, these disadvantages may not be offset by competitive advantages for ad networks that operate in a privacy-preserving manner.

From another perspective, our current study is limited to those ad libraries that are simply “piggybacked” into host apps. Particularly, current ad libraries are typically self-contained (as a standalone package) so that they can be readily included by app developers. However, it is possible to have more advanced mechanisms (e.g., collusion [129], re-delegation [93, 100], or indirect channels [151]) that could avoid using dangerous Android APIs being modeled by AdRisk while still accessing various personal information on the phone. Note that there are some ongoing research projects that aim to detect or mitigate these attacks [56, 57, 82, 93].
How to extend AdRisk to seamlessly integrate these systems remains an interesting task for future work.

5.6 Summary

In this chapter, we systematically examine the security and privacy issues raised by in-app ad libraries. We analyze 100 ad libraries selected from a sample of 100,000 apps collected from Google Play, and find that even among some of the most widely-deployed ad libraries, there exist threats to security and privacy. Such threats range from collecting unnecessarily intrusive user information to allowing third-party code of unknown provenance to execute within the hosting app. Since Android’s permissions model cannot distinguish between actions performed by an ad library and those performed by its hosting app, the current Android system provides little indication of the existence of these threats within any given app. In other words, architectural choices undermine the trust model behind Android’s permissions system, leading to privileges being reexposed to parties that are not explicitly trusted by the user.
Chapter 6

Conclusion and Future Work

In this dissertation, we first proposed, in Chapter 3, the use of virtualization techniques to defeat kernel code injection attacks by emulating a Harvard architecture on a modern x86 processor. The approach was shown to be more efficient than previous similar techniques, as it leveraged peculiarities of the underlying hardware, essentially stripping away the problematic von Neumann architecture of the machine for the purposes of kernel memory. Chapter 4 defined and demonstrated the threat of capability leaks in the Android framework by conducting a systematic study of Android devices from several major OEMs. Permission re-delegation attacks were found to be endemic to OEM-customized firmware, which spurred the research community to develop better systems to identify such problems, as well as motivating change within the OEM industry. Lastly, Chapter 5 presented a large-scale study of the nature of advertisement libraries in the Android ecosystem, concluding that their very embedded nature violates the trust model implicit in Android’s permission system. Even worse, these embedded libraries were shown to have abused the trust of users and developers by engaging in a variety of bad behaviors.

Cumulatively, these works have already informed and motivated a substantial amount of follow-on work, as shown in Chapter 2. We have shown that architectural choices cause privilege leakage in a number of different contexts, even within the limited scope of this dissertation. In the future, we propose carrying the same insights into a variety of new domains, including:

- **ARM TrustZone Operating Systems.** The ARM TrustZone extensions are designed for ease of implementation in hardware. TrustZone operating systems generally communicate with so-called “rich” operating systems, such as Android, by passing buffers back and forth; effectively the TrustZone is supposed to be a separate computer that happens to share the same processor and memory, rather than simply another privilege level. Unfortunately, TrustZone operating systems and rich operating systems are built upon very different principles, making it hard to authenticate and manage communication between
them. This situation could easily lead to privilege leakage, as TrustZone developers that do not understand the model of both the rich operating system and the TrustZone can fall prey to memory safety issues and capability leaks alike. This risk exists despite (and because of) the minimal contact between these two virtual “systems.” Studying the extent of any problems, as well as proposing mitigations that may improve the situation, would both be interesting avenues for further inquiry.

- **Hardware Organization.** As the world moves to ever-smaller computing devices, they are often fabricated as a unit. For example, there is great pressure in the smartphone industry to make each new model thinner, lighter, and more powerful than the last. Hardware engineers are therefore forced to make difficult decisions in order to conserve resources like space and battery power. This can lead to problems beyond the obvious: even if the device is made up of high-quality components with robust security features, if the greater system does not connect them in the right way, security problems can result. A systematic inquiry into the trust relationships between these components could have illuminating results: given the disconnects between the privilege-separation mechanisms supported by each subsystem in a highly-integrated, sophisticated device such as a smartphone, attackers may be able to leverage control of one subsystem to affect others.

- **The Internet of Things.** Small processors are being embedded into a wide variety of otherwise prosaic devices: light bulbs, door locks, toaster ovens, and so on. These devices are not capable of rich interaction with a user, and so must talk through one or more intermediaries to receive commands from them. Being highly constrained on cost and capability, these devices speak simple protocols that may lack strong authentication – potentially leading to “confused deputy” attacks that involve hardware components instead of software ones, as an untrusted device may be forwarding commands that did not originate from the user. Given their influence on the material world, developing ways to properly secure such devices, while respecting their constraints, is a promising direction for research.
REFERENCES


[27] Target says up to 70 million more customers were hit by December data breach. https://www.washingtonpost.com/business/economy/target-says-70-million-customers-were-hit-by-dec-data-breach-more-than-first-reported/2014/01/10/0ada1026-79fe-11e3-8963-b4b654bcc9b2_story.html.


[77] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege
Escalation Attacks on Android. In Proceedings of the 3th Information Security Conference,
October 2010.

[78] Drew Davidson, Matt Fredrikson, and Benjamin Livshits. Morepriv: Mobile os support
for application personalization and privacy. In Proceedings of the 30th Annual Computer
ACM.

[79] Soteris Demetriou, Nan Zhang, Yeonjoon Lee, Xiaofeng Wang, Carl Gunter, Xiaoyong
Zhou, and Michael Grace. HanFence: SDN-driven protection of smart home WiFi devices


[81] Lieven Desmet, Wouter Joosen, Fabio Massacci, Pieter Philippaerts, Frank Piessens, Ida
Siahaan, and Dries Vanoverberghe. Security-by-contract on the .NET platform. Informa-

[82] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. QUIRE:

[83] Joshua Drake, Zuk Avraham, and Nikias Bassen. Experts Found a Unicorn in the Heart

[84] G.W. Dunlap, S.T. King, S. Cinar, M.A. Basrai, and P.M. Chen. ReVirt: Enabling In-
trusion Analysis through Virtual-Machine Logging and Replay. ACM SIGOPS Operating

features/logs-and-services/loggers/cARRIERIQ/.

PrivacyLeaks in iOS Applications. In Proceedings of the 18th Annual Network and
Distributed System Security Symposium, NDSS ’11, February 2011.

McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for
Realtime Privacy Monitoring on Smartphones. In Proceedings of the 9th USENIX Sympos-
ium on Operating Systems Design and Implementation, OSDI ’10, pages 1–6, February
2010.

USENIX Security ’11, August 2011.


101


