ABSTRACT

WU, LEI. Vulnerability Detection and Mitigation in Commodity Android Devices. (Under the direction of Xuxian Jiang.)

The smartphone market has grown explosively in recent years, as more and more consumers are attracted to the sensor-studded multipurpose devices. However, with the popularity of Android platform, a large number of vulnerabilities have been observed and exploited in the wild.

In this dissertation, we aim to detect and quantify the presence of vulnerabilities in commodity Android devices, and propose corresponding mitigation solutions. In particular, we focus on two kinds of vulnerabilities, i.e., app-level vulnerabilities in pre-loaded apps and kernel-level vulnerabilities in device firmware images. To these ends, we first propose a system named SEFA to detect app-level vulnerabilities and evaluate their impact. Using SEFA, we quantitatively analyze ten representative stock Android images from five popular smartphone vendors (with two models from each vendor). Our evaluation results are worrisome: all examined devices are vulnerable. Our results also show that vendor customizations are significant on stock Android devices and on the whole responsible for the bulk of the security problems we detected in each device. To make matter worse, the slow update cycle makes the threats more serious, since it provides the attackers with more opportunities to launch the attack.

In addition, we design a system named KSEFA to detect kernel-level vulnerabilities, including known and unknown ones, and evaluate their impact. First of all, KSEFA is capable of quantitatively measuring the effectiveness of defense mechanisms by verifying whether the known vulnerabilities can actually be exploited to perform privilege escalation. Our evaluation results show that the known kernel-level vulnerabilities can significantly affect the security of Android devices, and the situation would remain severe even after several months of the disclosed dates. Second, based on the obtained insights, we are able to build KSEFA to discover certain zero-day vulnerabilities by performing a static disassembly-level data-flow analysis. In fact, several zero-day vulnerabilities have been discovered by using KSEFA, and these findings demonstrate the effectiveness and promise of the proposed system. Finally, we evaluate vendors’ response times to release the official patches. The evaluation results show that the responses can hardly be regarded as timely, and leave a wide open time window for exploitations.

Our work calls for community attention for effective solutions against the aforementioned threats. In light of this, we propose our approach, i.e., a novel transparent policy-based framework named DVMFA, to protect the vulnerable devices. DVMFA is a context-aware system which is capable of patching app- and kernel-level vulnerabilities dynamically in a fine-grained manner. Specifically, we first design a policy language for kernel-level vulnerabilities as the base set, and then extend it to support app-level vulnerabilities with richer semantic specification and bindings. Our evaluation results show that the vulnerabilities we studied can be effectively prevented by DVMFA with negligible performance overhead.
Vulnerability Detection and Mitigation in Commodity Android Devices

by
Lei Wu

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
2015

APPROVED BY:

______________________________  ______________________________
Vincent W. Freeh              Xipeng Shen

______________________________  ______________________________
Huiyang Zhou                 Xuxian Jiang
Chair of Advisory Committee
DEDICATION

To my parents, wife and daughter.
BIOGRAPHY

Lei Wu is originally from Huai’an, China. He received his Bachelor of Engineering degree in Software Engineering and his Master of Engineering degree in Computer Science from Nanjing University in China in 2005 and 2008, respectively. Since 2011, he spent four years in North Carolina State University (NCSU) to pursue his Doctor of Philosophy degree in Computer Science under the direction of Dr. Xuxian Jiang. His research interests lie primarily in mobile security such as the vulnerability detection and mitigation in commodity Android device (the focus of this dissertation). He is expected to graduate with a Ph.D. degree in Computer Science from NCSU in August 2015, and join Qihoo 360 Technology Co. Ltd. in Fall 2015.
ACKNOWLEDGEMENTS

The pursuit and completion of the Ph.D. degree not only requires one’s passion and persistence, but also asks for encouragement and support from his or her surroundings. I would like to express my gratitude to those who have supported and encouraged me with their invaluable help at all stages of my graduate study.

First of all, I would like to thank my adviser, Prof. Xuxian Jiang, for his important support and insightful guidance during my graduate study. His high standard of research and great enthusiasm in the work significantly benefits and motivates me. I am also thankful to Prof. Vincent W. Freeh, Prof. Xipeng Shen, and Prof. Huiyang Zhou for their time and efforts serving in my PhD thesis committee and providing valuable advice that helped to improve this dissertation. Second, I am grateful to all my coauthors for their helpful discussions and collaborations, i.e., my colleagues in the Cyber Defense Lab at NCSU: Michael Grace, Yajin Zhou and Chiachih Wu. Third, I would like to give my thanks to my friends for their help during my Ph.D. study: Zhi Wang, Wu Zhou, Kunal Patel, Feifei Wang, Qiang Zhang, Xiaocheng Zou, Yang Liu, Xiaoqing Luo, Pu Yang and Wenbo Shen.

In addition to academics, I received help in many other perspectives as well: Xuxian Jiang and Huiyang Zhou from NCSU, Cliff Zou, Ping Wang, Baber Aslam and Yuqing Lin from University of Central Florida, Men Long, Hormuzd M. Khosravi and Prashant Dewan from Intel Research Lab, provided valuable help with constructive feedbacks and advice; Carol Allen, Margery Page, Tom Ryan, Douglas Reeves, George Rouskas, and other staff members from the Department of Computer Science and the Office of International Services also deserve my gratitude for their excellent work. I am also grateful to organizations, i.e., National Science Foundation (NSF) and Army Research Office (ARO), for their generous funding support for my graduate research. As usual, any opinions, findings, and conclusions or recommendations expressed in this dissertation do not necessarily reflect the views of the NSF and the ARO.

Last but certainly not least, I deeply appreciate the everlasting support and encouragement from my family, particularly my wife, Feiye Wang, and our daughter Jinyi. Without their continuous support, none of this would have been possible, not to mention enjoyable. Also, I can never overemphasize the importance of my parents in shaping who I am and giving me the gift of education.
# TABLE OF CONTENTS

## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Tables</td>
<td>vii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>viii</td>
</tr>
</tbody>
</table>

## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 1: Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Chapter 2: Related Work</td>
<td>7</td>
</tr>
<tr>
<td>Chapter 3: Detection of App-Level Vulnerabilities</td>
<td>11</td>
</tr>
<tr>
<td>Chapter 4: Detection of Kernel-Level Vulnerabilities</td>
<td>36</td>
</tr>
</tbody>
</table>

## Chapter 1: Introduction

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Background and Problem Overview</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Our Approach</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Dissertation Contributions</td>
<td>4</td>
</tr>
<tr>
<td>1.4 Terminology</td>
<td>5</td>
</tr>
<tr>
<td>1.5 Dissertation Organization</td>
<td>6</td>
</tr>
</tbody>
</table>

## Chapter 2: Related Work

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Vulnerability Detection</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Vulnerability Mitigation</td>
<td>8</td>
</tr>
<tr>
<td>2.3 Android Security in General</td>
<td>9</td>
</tr>
</tbody>
</table>

## Chapter 3: Detection of App-Level Vulnerabilities

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Introduction</td>
<td>11</td>
</tr>
<tr>
<td>3.2 Design</td>
<td>13</td>
</tr>
<tr>
<td>3.2.1 Provenance Analysis</td>
<td>14</td>
</tr>
<tr>
<td>3.2.2 Permission Usage Analysis</td>
<td>15</td>
</tr>
<tr>
<td>3.2.3 Vulnerability Analysis</td>
<td>17</td>
</tr>
<tr>
<td>3.3 Implementation and Evaluation</td>
<td>22</td>
</tr>
<tr>
<td>3.3.1 Provenance Analysis</td>
<td>22</td>
</tr>
<tr>
<td>3.3.2 Permission Usage Analysis</td>
<td>24</td>
</tr>
<tr>
<td>3.3.3 Vulnerability Analysis</td>
<td>27</td>
</tr>
<tr>
<td>3.4 Discussion</td>
<td>34</td>
</tr>
<tr>
<td>3.5 Summary</td>
<td>35</td>
</tr>
</tbody>
</table>

## Chapter 4: Detection of Kernel-Level Vulnerabilities

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Introduction</td>
<td>36</td>
</tr>
<tr>
<td>4.2 Design</td>
<td>38</td>
</tr>
<tr>
<td>4.2.1 Preprocessing Firmware Images</td>
<td>39</td>
</tr>
<tr>
<td>4.2.2 Verifying Known Vulnerabilities</td>
<td>40</td>
</tr>
<tr>
<td>4.2.3 Discovering Unknown Vulnerabilities</td>
<td>44</td>
</tr>
<tr>
<td>4.3 Implementation and Evaluation</td>
<td>46</td>
</tr>
<tr>
<td>4.3.1 Evaluation Preparation</td>
<td>46</td>
</tr>
<tr>
<td>4.3.2 Verifying Known Vulnerabilities</td>
<td>48</td>
</tr>
<tr>
<td>4.3.3 Discovering Unknown Vulnerabilities</td>
<td>53</td>
</tr>
<tr>
<td>4.3.4 Response Time</td>
<td>54</td>
</tr>
<tr>
<td>4.4 Discussion</td>
<td>57</td>
</tr>
<tr>
<td>4.5 Summary</td>
<td>58</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

| Table 3.1 | Provenance analysis of representative devices | 23 |
| Table 3.2 | Release dates of examined Android devices | 24 |
| Table 3.3 | Permission usage analysis of representative devices | 25 |
| Table 3.4 | Vulnerability analysis of representative devices (I) | 28 |
| Table 3.5 | Vulnerability analysis of representative devices (II) | 30 |
| Table 3.6 | Classification of detected vulnerabilities | 31 |
| Table 3.7 | Distribution of vulnerabilities among examined devices | 32 |
| Table 4.1 | Escalating targets | 41 |
| Table 4.2 | Patterns of manufacturer-specific vulnerabilities | 45 |
| Table 4.3 | Target device models | 48 |
| Table 4.4 | Overall results with two observation points | 49 |
| Table 4.5 | Measurement of effectiveness | 53 |
| Table 4.6 | Vendor-patch time | 57 |
| Table 5.1 | Patchable vulnerabilities of current DVMFA | 68 |
| Table 5.2 | Portability of DVMFA | 69 |
| Table A.1 | Summary of collected vulnerabilities | 88 |
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1</td>
<td>Three key components of the dissertation</td>
<td>2</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>The overall architecture of SEF A</td>
<td>13</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Distributions of overprivileged apps</td>
<td>26</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Vulnerabilities due to vendor customizations</td>
<td>29</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>Vendor-specific vulnerability evolution</td>
<td>31</td>
</tr>
<tr>
<td>Figure 3.5</td>
<td>Distribution of cross-app vulnerabilities</td>
<td>32</td>
</tr>
<tr>
<td>Figure 3.6</td>
<td>An example path of reflection attack</td>
<td>33</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>The overall architecture of KSEFA</td>
<td>39</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>Device properties of Google’s Nexus 4</td>
<td>40</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>An example of the flow network</td>
<td>44</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>Flow of discovering unknown vulnerabilities</td>
<td>46</td>
</tr>
<tr>
<td>Figure 4.5</td>
<td>Vulnerable firmware images by vulnerabilities</td>
<td>50</td>
</tr>
<tr>
<td>Figure 4.6</td>
<td>Vulnerable firmware images by device vendors</td>
<td>51</td>
</tr>
<tr>
<td>Figure 4.7</td>
<td>Nexus 4 and CVE-2013-6123</td>
<td>52</td>
</tr>
<tr>
<td>Figure 4.8</td>
<td>Snippet of execution result of /dev/camera-fdvt</td>
<td>54</td>
</tr>
<tr>
<td>Figure 4.9</td>
<td>Snippet of execution result of /dev/camera-isp</td>
<td>55</td>
</tr>
<tr>
<td>Figure 4.10</td>
<td>Community-patch time and vendor-patch time</td>
<td>56</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>The overall architecture of DVMFA</td>
<td>61</td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>Flow of sending intent</td>
<td>65</td>
</tr>
<tr>
<td>Figure 5.3</td>
<td>Screenshots of interception white list and warning message</td>
<td>70</td>
</tr>
<tr>
<td>Figure 5.4</td>
<td>Logs of blocking intent and uri</td>
<td>71</td>
</tr>
<tr>
<td>Figure 5.5</td>
<td>Screenshots of rebooting system explicitly</td>
<td>72</td>
</tr>
<tr>
<td>Figure 5.6</td>
<td>CVE-2014-3153: exploitation and mitigation</td>
<td>73</td>
</tr>
<tr>
<td>Figure 5.7</td>
<td>Log of blocking CVE-2014-3153</td>
<td>73</td>
</tr>
<tr>
<td>Figure 5.8</td>
<td>Benchmark measurement of DVMFA</td>
<td>74</td>
</tr>
<tr>
<td>Figure 5.9</td>
<td>AnTuTu measurement results of DVMFA</td>
<td>74</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Background and Problem Overview

The smartphone market has grown explosively in recent years, as more and more consumers are attracted to these multipurpose devices. According to IDC [49], smartphone vendors shipped a total of 1.3 billion mobile phones in 2014. Android has continued to dominate the worldwide market, i.e., capturing approximately 81.5% of the global smartphone market share in year 2014, compared to approximately 78.7% in 2013. Further, the number of Android apps has increased substantially with the growth of third-party developers. It was reported that [4] by the end of 2014, Android had already surpassed iOS in terms of the number of apps available in their own official markets.

However, with the popularity of the Android platform, many vulnerabilities that may derive from different layers or components in Android devices have been observed and exploited in the wild. For example, security flaws in pre-loaded apps have been observed in previous work [40], while vulnerabilities derived from the Android framework and Linux kernel have also been investigated by researchers, e.g., [9, 59, 92, 99]. Unfortunately, those studies were still too ad-hoc to evaluate the overall impact on security of the vulnerabilities quantitatively.

In this dissertation, we aim to detect and quantify the presence of vulnerabilities that can actually be exploited in commodity Android devices, and propose corresponding mitigation solutions. In particular, we focus on two kinds of vulnerabilities, i.e., app-level vulnerabilities in pre-loaded apps \(^1\), and kernel-level vulnerabilities in device firmware images. Specifically, we envision a systematic approach in this dissertation to: 1) evaluate the impact of the vulnerabilities on security, and 2) mitigate the threats introduced by those vulnerabilities. Figure 1.1 depicts the three key components of the dissertation, i.e.,

---

\(^1\)Unless specified, otherwise, app-level vulnerabilities mentioned in the following refer to vulnerabilities in pre-loaded apps, which are more privileged than others, and may cause greater damage if they are vulnerable.
a Security Evaluation Framework for Android (SEFA, Chapter 3), a Kernel-Level Security Evaluation Framework for Android (KSEFA, Chapter 4) and a Dynamic Vulnerability Mitigation Framework for Android (DVMFA, Chapter 5). Note that SEFA and KSEFA focus on the detection of app-level and kernel-level vulnerabilities, respectively. For a given device, the detection in this context includes the following two aspects:

- **Detection of known vulnerabilities:** We will determine the presence of known vulnerabilities and then verify whether or not they are exploitable in the device;

- **Detection of unknown vulnerabilities:** Our goal is to discover unknown vulnerabilities that are exploitable in the device.

As demonstrated in the following chapters (e.g., Chapter 4), the insights obtained from the detection of known vulnerabilities can be used to facilitate the detection of those that are unknown.

### 1.2 Our Approach

We propose SEFA to detect app-level vulnerabilities in commodity Android devices. Given an Android device with a target firmware image ², SEFA first preprocesses it and imports a variety of information about the image into a local database, including the number of apps and numerous information about

²Unless specified, otherwise, *image* and *firmware image* are interchangeable in the following.
each app, such as the list of requested permissions, declared components, and the set of Android APIs used. Then, SEFA compares each pre-loaded app with various apps in the original AOSP to determine its source and further performs a system-wide data-flow analysis to detect possible vulnerabilities. In our study, we have applied SEFA to ten flagship phone models from five popular vendors: Google, Samsung, HTC, LG, and Sony. For each vendor, we selected two phones: one from the current crop of Android 4.x phones, and one from the previous generation of 2.x devices. This slate of devices allows us to conduct two comparative analyses: horizontal differential analysis, which compares the various manufacturers’ offerings for a given generation, and vertical differential analysis, which studies the evolution of any given vendor’s security practices chronologically. Our results are worrisome: all devices examined are vulnerable. The results also show that vendor customizations are significant on stock Android devices and, on the whole, are responsible for a large proportion (more than 60%) of the vulnerabilities in each device. Importantly, the slow update cycle makes the threats more serious, as it offers more opportunities to launch attacks.

We next propose KSEFA to detect kernel-level vulnerabilities in commodity Android devices. Given an Android device with a target firmware image, KSEFA first extracts necessary information from the device, and then performs two in-depth analyses to detect possible vulnerabilities. Firstly, KSEFA is capable of measuring the effectiveness of defense mechanisms quantitatively by verifying whether or not the known vulnerabilities can actually be exploited to perform privilege escalation. Specifically, we selected eight known kernel-level vulnerabilities, and nine typical devices from three representative vendors (i.e., Google, Samsung, and HTC). We chose three phone models for each vendor, and collected their updated firmware images to perform the evaluation. Our empirical results show that the known kernel-level vulnerabilities can affect the security of Android devices significantly. In particular, we show that a newly publicly disclosed kernel-level vulnerability could be leveraged to compromise approximately 85% devices (with the latest images), while half of them, even with the latest updated images, were still vulnerable three months later. Our results also demonstrate that the defense mechanisms deployed in Android devices are not as effective as expected. Secondly, based on the insights obtained, we are able to build KSEFA to discover certain zero-day vulnerabilities by performing a static disassembly-level data-flow analysis. In fact, several zero-day vulnerabilities have been discovered using KSEFA, and these findings demonstrate the effectiveness and promise of the proposed system. Moreover, we evaluate vendors’ response times to release official patches. The evaluation results show that the responses are not timely (230 days on average), and leave a wide open time window for exploitations.

Our work calls for community attention for effective solutions against the aforementioned threats. In light of this, we propose our approach, i.e., a novel transparent policy-based framework named DVMFA, to protect the vulnerable devices. DVMFA is a context-aware system that is capable of patching app- and kernel-level vulnerabilities dynamically in a fine-grained manner. Specifically, we first design a policy
language for kernel-level vulnerabilities as the base set, and then extend it to support app-level vulnerabilities with richer semantic specification and bindings. However, we perform the patching differently: app-level vulnerabilities are patched by intercepting corresponding Android APIs with an app-level monitoring module, while kernel-level vulnerabilities are patched by intercepting related system calls with a loadable kernel module. Our evaluation results show that the vulnerabilities we studied can be effectively prevented by DVMFA with negligible performance overhead.

1.3 Dissertation Contributions

The contributions of this dissertation are summarized as follows:

• **The systematic study of app-level vulnerabilities in commodity Android devices**  We designed and implemented SEFA, a framework able to evaluate quantitatively the impact on security of app-level vulnerabilities in Android devices. This tool performs several analyses to study the provenance, permission usage and vulnerability distribution of the pre-loaded apps that constitute a device’s firmware image. The results showed that all devices examined were vulnerable, and more than 60% of the vulnerabilities identified stemmed from the vendors’ modifications to the firmware. Further, for most of the manufacturers in our study, these patterns were stable over time, which highlights the need for a heightened focus on security by the smartphone industry.

• **The systematic study of kernel-level vulnerabilities in commodity Android devices**  We designed and implemented KSEFA, a framework able to evaluate quantitatively the impact on security of kernel-level vulnerabilities in Android devices. First, KSEFA is capable of measuring the effectiveness of defense mechanisms by verifying whether or not the known vulnerabilities can actually be exploited to perform privilege escalation. Our empirical results showed that the kernel-level vulnerabilities affect the security of Android devices significantly. On average, 84.78% of the latest firmware images were vulnerable to a newly publicly disclosed vulnerability, and 47.83% of the latest updated images were still vulnerable after three months. Second, based on the insights obtained, we were able to build KSEFA to discover certain zero-day vulnerabilities by performing a static disassembly-level data-flow analysis. Several zero-day vulnerabilities were discovered, and these findings demonstrated the effectiveness and promise of the proposed system.

• **Mitigation of app- and kernel-level vulnerabilities in commodity Android devices**  We designed and implemented DVMFA, a transparent, policy-based framework, to mitigate the aforementioned threats. DVMFA is a context-aware system that is capable of patching app- and kernel-level vulnerabilities dynamically in a fine-grained manner. It also does not require any modifica-
tion or recompilation of pre-loaded apps or device firmware images. Our evaluation showed that the vulnerabilities we studied can be prevented effectively by DVMFA with negligible performance overhead (with less than 3% slowdown).

1.4 Terminology

This section establishes the terminology that is used throughout this dissertation.

- **Chipset Manufacturer**  A manufacturer of System-on-Chip (SoC), CPU or GPU, e.g., Qualcomm.

- **Known Vulnerability**  A vulnerability that has already been disclosed publicly.

- **Unknown Vulnerability**  A vulnerability that has not been disclosed, i.e., zero-day vulnerability. In comparison with the term known vulnerability, we prefer to use unknown vulnerability rather than zero-day vulnerability in the following chapters, but they are interchangeable.

- **Vulnerability Detection**  The detection of known vulnerabilities include a two-step activity, first determining the presence of the vulnerabilities and then verifying whether or not they are exploitable in a given device; the detection of unknown vulnerabilities refers to the discovery of vulnerabilities that are exploitable in the device.

- **App-level Vulnerabilities**  Vulnerabilities derived from pre-loaded apps in a commodity Android device. In particular, this type of vulnerabilities can be used to either launch permission re-delegation attacks [32] or lead to content leaks [101], which will be explained in Chapter 3.

- **Framework-level Vulnerabilities**  Vulnerabilities derived from application framework, Android runtime or libraries in a commodity Android device.

- **Kernel-level Vulnerabilities**  Vulnerabilities derived from the Linux kernel directly or the device driver provided by chipset manufacturers, therefore we call them kernel-generic and manufacturer-specific, respectively. These types of vulnerabilities can be exploited to gain root privilege, which will be discussed in Chapter 4.

- **Disclosed Date**  The date when a vulnerability is disclosed to the public (or community).

- **Community-patch Date**  The date when the Linux kernel development community or chipset manufacturers release a patch to fix a vulnerability of which they are in charge.
• **Community-patch Time** The time window required for the Linux kernel development community or chipset manufacturers to fix vulnerabilities they are in charge. This is equivalent to the community-patch date subtracted from the disclosed date. Without ambiguity, we simply use the community-patch time to refer to the time window.

• **Vendor-patch Date** The date when device vendors (e.g., HTC) or carriers (e.g., AT&T) release the corresponding updated firmware images after the community-patch date of a vulnerability.

• **Vendor-patch Time** The time window required for device vendors or carriers to release the corresponding over-the-air (OTA) updates after the community-patch date of a vulnerability. Without ambiguity, we simply use the vendor-patch time to refer to the time window.

• **Patch Time** The total time taken to release the corresponding updated firmware images after a vulnerability is disclosed. This is the sum of community-patch time and vendor-patch time.

### 1.5 Dissertation Organization

The remainder of this dissertation is organized as follows. Firstly, we present closely related work about the vulnerability detection and mitigation in the Android platform in Chapter 2. Secondly, we describe the design, implementation, and evaluation of SEFA, KSEFA, and DVMFA in Chapters 3, 4, and 5, respectively. Finally, we conclude the dissertation and discuss future work in Chapter 6.
Chapter 2

Related Work

In this chapter, we summarize closely related work in the areas of vulnerability detection and mitigation, and Android security in general.

2.1 Vulnerability Detection

Related work in this section includes research efforts relevant to mobile devices.

**Permission re-delegation attack**  Permission re-delegation attack, a form of the classic confused-deputy attack [44], has been known for some time to be a problem on the Android platform. ComDroid [16], Woodpecker [40], and CHEX [55] all apply static analysis techniques to find vulnerabilities in either third-party or pre-loaded apps. SEFA is most similar to the latter two systems. However, our system also performs provenance analysis, allowing us to determine the impact of vendor customizations on security. Note that ComDroid and Woodpecker mainly focus on in-component vulnerability detection, CHEX can additionally detect cross-component ones. Our work is the most comprehensive yet, as it can find in-component, cross-component, and cross-app vulnerabilities. Specifically, cross-app vulnerabilities account for 8.90% of the vulnerabilities that we found, a significant proportion that also leads to similar, if not greater, security risks.

**Root level privilege escalation attack**  Vulnerabilities that can be used to perform privilege escalation to gain root access in Android devices, have been discovered and exploited in the wild [59]. Hbarth et al. [48] proposed a framework to achieve permanent privilege escalation. X-Ray [90] is a tool that is able to detect the presence of eight publicly known vulnerabilities released before 2013. Researchers [59] have also tracked the detection rate fluctuations of X-Ray over time, which has been decreased from 60.6% in late 2012 to 13.14% in early 2014. Their results show that many devices were still vulnerable to old known vulnerabilities.
Vulnerability analysis ContentScope [101] tries to identify vulnerabilities related to third-party, unprotected content providers. Our own work uses a similar concept, but is concerned with content providers in pre-loaded apps. Peril [99] reveals some security risks by observing that the permissions of the device driver related files were not set properly. SecUP [92] is able to detect a type of vulnerability named pileup flaw, which can be exploited when performing OTA updates.

Malware analysis Several works have attempted to survey the landscape of the malware on Android (e.g., MalGenome [100]) as well as general apps [31, 71]. Other works, such as DroidRanger [102], RiskRanker [41], Peng et al. [67] and MAST [14] all have been concerned with finding malicious apps in app markets that contain a large number of benign apps. DroidScope [94] uses virtualization to perform semantic view reconstruction, much like a number of other desktop systems, to analyze Android malware. The insights developed in these works were useful in informing our own about the potential dangers of malicious third-party apps.

2.2 Vulnerability Mitigation

Related work in this section includes research efforts to mitigate the threats introduced by vulnerabilities.

Defense mechanisms for desktop PCs Linux kernel exploitation and corresponding defense mechanisms have been discussed and studied for decades [63, 64]. A number of defense mechanisms, e.g., kernel Address Space Layout Randomization (ASLR), have been proposed and adopted by the community [52]. Although they may be bypassed with advanced attack techniques, these defense mechanisms do make it more difficult for attackers to accomplish their goals. In addition, some approaches have been proposed to patch kernels automatically. For example, KSplice [5] is capable of applying patches (if without significant semantic changes to kernel data structures) for desktop PCs at runtime without rebooting the OS. Such patching systems can also be used to mitigate threats.

Unfortunately, for various reasons, many defense mechanisms have not yet been adopted by Android. Tamer et al. [82] has summarized the Android Security Enhancements adopted by the AOSP, including both user space and kernel space protection features. Some framework-level vulnerabilities, e.g., system service, which can also be used to perform exploitation and gain root privilege, may be prevented already by user space defense mechanisms. However, it may not come as a surprise that most of them are not feasible for preventing kernel-level vulnerabilities.

Defense solutions for smartphone platforms Considering the specific nature of smartphone platforms, several systems, including TaintDroid [30], PiOS [28], Apex [61], MockDroid [8], TISSA [103], AppFence [46], Aurasium [93], SORBET [34] and CleanOS [83] have been proposed to study privacy leak issues. They all try to protect (or mitigate) the privacy leak problem by adjusting or improving the
core smartphone framework. Some other systems [12, 27, 32] are designed to mitigate the permission re-delegation problem by either checking IPC call chains or by monitoring the run-time communication between apps. RBGDroid [65] is proposed to prevent an malicious app from acquiring root-level privilege and accessing protected resource. MoCFI [25] implements a control-flow integrity enforcement framework for apps on iOS. Other works have tried to protect security in a different manner. For example, virtualization techniques are leveraged by Cells [2] and L4Android [54]. AirBag [88] is a lightweight, OS-level virtualization-based system that can be used to isolate untrusted Android apps, while SEAndorid [80] provides Mandatory Access Control (MAC), and has been adopted by the AOSP since Android 4.2. FlaskDroid [11] is capable of providing fine-grained MAC policies based on SEAndroid.

Additionally, research work has been done to patch the vulnerable Android devices. PatchDroid [60] is a system that can be used to patch framework-level vulnerabilities. For example, AppSealer [96] is designed to generate security patches for component hijacking vulnerabilities of Android apps. However, neither of them covers both app- and kernel-level vulnerabilities. FireDroid [74] is a policy-based security framework and is capable of controlling the behaviour of malicious apps by intercepting system calls using the ptrace mechanism. Although it is able to patch vulnerabilities dynamically, some limitations (e.g., the requirement to modify the firmware image) decrease the feasibility of its deployment.

### 2.3 Android Security in General

**App provenance analysis**  Provenance provides an important context that can be used to evaluate the results of our other analyses. However, how to determine the provenance of a mobile app or a piece of code is a well-known problem that has stimulated much research. For example, traditional similarity measurement has been used widely in malware (e.g., virus and worm) clustering and investigation. Kruegel et al. [53] proposed an approach to detect polymorphic worms by identifying structural similarities between control flow graphs. SMIT [47], a malware database management system based on malware’s function-call graphs, is designed to perform large scale malware indexing and queries. Note that these approaches were developed primarily for PC malware and are considerably sophisticated and complex than our own. In this dissertation, we use chains of method signatures to determine whether a pre-loaded app is, in fact, an altered version of legitimate one. And previous insights are instrumental in improving the accuracy and completeness of our approach.

In the Android platform, one line of inquiry concerns the detection of repackaged apps, those apps that have been altered and re-uploaded by some party other than the original author. DroidMOSS [98], DNADroid [18] and PiggyApp [97] all focus on the detection of repackaged apps in Android app markets, while AppInk [89] focuses on deterring repackaging behavior with watermarking techniques.
These efforts all must establish the ancestry of any given app, which mirrors our efforts to understand, longitudinally, the evolution of firmware images using vertical differential analysis (Chapter 3).

Another line of research is connected with mobile advertisement libraries, as they live within other apps and share their permissions. They have been demonstrated to be another important source of privacy leaks [39]; furthermore, Book et al.’s longitudinal study [10] shows that negative behaviors may be growing more common in such libraries over time. As a result, several mitigation measures have been proposed. For example, new APIs and permissions [66] might be added to attempt to isolate such alien code; AdSplit [78], in contrast, moves advertisement code into another process to allow the core Android framework to issue it different permissions, and thus enforces a different policy. Our work has a strange kinship with these works, in that we similarly are interested in poorly-tagged vendor customizations mixed in with code from other sources. While we operate on a different scale to evaluate whole phone images instead of individual third-party apps, many of the same concepts apply. Furthermore, there similarly exists a disconnection between the trust a user may afford the open-source, heavily vetted AOSP and the vendor’s customizations of it – attempting to mitigate the flaws introduced by the vendor would be an interesting topic for future work.

**App permission usage analysis** Our permission usage analysis is built upon the accumulated insight of a number of other works in this area. For example, Stowaway [33], Vidas et al. [85] and PScout [6] have all studied the problem of overprivileged third-party apps and provide permission mappings in various ways. Barrera et al. [7] studied the permission usage patterns of third-party apps by applying self-organizing maps. However, none of these works has analyzed the problem of permission overprivilege in pre-loaded firmware apps, which is one key focus of this dissertation.

Others have attempted to infer certain security-related properties about apps based solely on their requested permissions. Kirin [29], for example, looks for hard-coded dangerous combinations of permissions to warn the user about potential malware. Sarma et al. [76], Peng et al. [67] and Chakradeo et al. [14], on the other hand, used machine learning techniques to automatically classify apps as potentially malicious based on the permissions they request. In our study, we do not attempt to look for such emergent effects in the permissions requested by a pre-loaded app as the examined stock images are normally released by reputable and trustworthy entities.
Chapter 3

Detection of App-Level Vulnerabilities

In this chapter, we focus on the detection of app-level vulnerabilities in commodity Android devices.

3.1 Introduction

Android’s popularity is due in part to it being an open platform. Google produces a baseline version of Android, then makes it freely available in the form of the Android Open Source Project (AOSP). Manufacturers and carriers are free to build upon this baseline, adding custom features in a bid to differentiate their products from their competitors. These customizations have grown increasingly sophisticated over time, as the hardware has grown more capable and the vendors more adept at working with the Android framework. Flagship devices today often offer a substantially different look and feel, along with a plethora of pre-loaded third-party apps.

From another perspective, vendor customizations will inherently impact overall Android security. Past work [40] has anecdotally shown that Android devices had security flaws shipped in their pre-loaded apps. Note that stock images include code from potentially many sources: the AOSP itself, the vendor, and any third-party apps that are bundled by the vendor or carrier. It is therefore important to attribute each particular security issue back to its source for possible bug-fixes or improvements.

In this chapter, we study app-level vulnerabilities in commodity Android devices and evaluate their impact. Especially, we intend to determine the source of the security issues that trouble Android smartphone images, then further determine how the situation is evolving over time. To that end, we develop a three-stage process to evaluate a given smartphone’s stock firmware image. First, we perform provenance analysis, aiming to classify each pre-loaded app into three categories: apps originating from the AOSP, apps customized or written by the vendor, and third-party apps that are simply bundled into the stock image. We then analyze, in two different ways, the security implications of each app: (1) Per-
mission usage analysis compares the permissions requested by the app with those that it actually uses, looking for apps that request more permissions than they use. This situation is known as permission overprivilege, and it indicates a poor understanding of the Android security model; (2) Vulnerability analysis, in comparison, looks for two general types of actual security vulnerabilities: permission re-delegation attacks and content leaks. Permission re-delegation attacks allow unprivileged apps to act as though they have certain sensitive permissions, while content leaks allow such apps to gain (unauthorized) access to private data.

To facilitate our analysis, we implement a Security Evaluation Framework for Android called SEFA to evaluate stock smartphone images. Given a particular phone firmware image, SEFA first preprocesses it and imports into a local database a variety of information about the image, including the number of apps and numerous information about each app, such as the list of requested permissions, declared components, and the set of used Android APIs. Then SEFA compares each pre-loaded app with various ones in the original AOSP to determine its source and further performs a system-wide data-flow analysis to detect possible vulnerabilities. In our study, we have applied SEFA to ten flagship phone models from five popular vendors: Google, Samsung, HTC, LG, and Sony. For each vendor, we selected two phones: one from the current crop of Android 4.x phones, and one from the previous generation of 2.x devices. This slate of devices allows us to do two comparative analyses: horizontal differential analysis compares the various manufacturers’ offerings for a given generation, while vertical differential analysis studies the evolution of any given vendor’s security practices chronologically.

Our evaluation results show that more than 81.78% of pre-loaded apps (or 76.34% of LOC) on stock Android devices are due to vendor customizations. It is worrisome to notice that all examined devices are vulnerable and vendor customizations were, on the whole, responsible for the bulk of the security problems suffered by each device. On average, 85.78% of all pre-loaded apps in examined stock images are overprivileged with a majority of them directly from vendor customizations. And vendor apps consistently exhibited permission overprivilege, regardless of generation. Our results also show that vendor customizations are responsible for a large proportion of the vulnerabilities in each phone. For the Samsung, HTC, and LG phones, between 64.71% and 85.00% of the vulnerabilities were due to vendor customizations. This pattern was largely stable over time, with the notable exception of HTC, whose current offering is markedly more secure than the last-generation model we evaluated.

The rest of this chapter is organized as follows. We present our methodology and system framework in Section 3.2, and describe implementation and evaluation results with case studies in Section 3.3. We then discuss for possible improvements in Section 3.4. Finally, we summarize this chapter in Section 3.5.
3.2 Design

Our goal here is to study vendor customizations on stock Android devices and assess corresponding security impact. Note that the software stack running in these devices are complex, and their firmware is essentially a collaborative effort, rather than the work of a single vendor. Therefore, we need to categorize the code contained in a stock image based on its authorship and audit it for possible security issues. After that, we can attribute the findings of the security analyses to the responsible party, allowing us to better understand the state of smartphone security practices in the industry and spot any evident trends over time.

Preprocessing
Provenance Analysis
Permission Usage Analysis
Vulnerability Analysis

Figure 3.1 The overall architecture of SEFA

In Figure 3.1, we summarize the overall architecture of the proposed SEFA system. Our system takes a stock phone image as its input, preprocessing each app and importing the results into a database. This database, initially populated with a rich set of information about pre-loaded apps (including information from their manifest files, signing certificates, as well as their code, etc.), is then used by a set of subsequent analyses. Each analysis reads from the database, performs its analysis, and stores its findings in the database.

To study the impact of vendor customizations on the security of stock Android smartphones, we have developed three such analyses. First, to classify each app based on its presumed authorship, we perform provenance analysis (Section 3.2.1). This analysis is helpful to measure how much of the baseline AOSP is still retained and how much customizations have been made to include vendor-specific features or third-party apps. To further get a sense of the security and privacy problems posed by each app, we use two different analyses: permission usage analysis (Section 3.2.2) assesses whether an app requests more permissions than it uses, while vulnerability analysis (Section 3.2.3) scans the entire image for concrete
security vulnerabilities that could compromise the device and cause damage to the user. Ultimately, by correlating the results of the security analyses with the provenance information we collected, we can effectively measure the impact of vendor customizations.

### 3.2.1 Provenance Analysis

The main purpose of provenance analysis is to study the distribution of pre-loaded apps and better understand the customization level by vendors on stock devices. Specifically, we classify pre-loaded apps into three categories:

- **AOSP app**: the first category contains apps that exist in the AOSP and may (or may not) be customized by the vendor.

- **vendor app**: the second category contains apps that do not exist in the AOSP and were developed by the vendor.

- **third-party app**: the last category contains apps that do not exist in the AOSP and were not developed by the vendor.

The idea to classify pre-loaded apps into the above three categories is as follows. First we collect AOSP app candidates by searching the AOSP, then we exclude these AOSP apps from the pre-loaded ones. After that, we can classify the remaining apps by examining their signatures (i.e., information in their certificate files) based on a basic assumption: third-party apps shall be private and will not be modified by vendors. Therefore, they will not share the same signing certificates with vendor apps.

In practice, this process is however not trivial. Since AOSP apps may well be customized by vendors, their signatures are likely to be changed as well. Although in many cases, the app names, package names or component names are unchanged, there do exist exceptions. For example, Sony’s *Conversations* app, with package name `com.sonyericsson.conversations`, is actually a customized version of the AOSP *Mms* app named `com.android.mms`. In order to solve this problem, we perform a call graph similarity analysis, which has been demonstrated to be an effective technique even to assist malware clustering and provenance identification [47].

To generate the call graph required by any such analysis, we add all method calls that can be reached starting from any *entrypoint* method accessible to other apps or the framework itself. However, we are hesitant to use graph isomorphism techniques to compare these call graphs, as they are complex and have undesirable performance characteristics. Instead, we notice that later analysis (Section 3.2.3) will use *paths*, sequences of methods that start at an entrypoint and flow into a *sink* (i.e., API or field which may require sensitive permissions, lead to dangerous operations or meet other special needs). Therefore,
we choose to preprocess each app, extract and compare the resulting paths, a much more straightforward process that still compare the parts of each app that we are most concerned with.

From our prototype, we observe that such a path-based similarity analysis is implementation-friendly and effective. Particularly, we use the return type and parameters (number, position and type) of each node (method) in the path as its signature. If the similarity between two paths exceeds a certain threshold, we consider these two paths are matching. And the similarity between two apps is largely measured based on the number of matched paths. In our prototype, to determine which apps belong to the AOSP, we accordingly take the approach: (1) by matching app names and package names; (2) by matching component names in the manifest file; (3) and then by calculating the similarity between paths and apps. We point out that a final manual verification is always performed to guarantee the correctness of the classification, which can also confirm the effectiveness of our heuristics.

During this stage, we also collect one more piece of information: the code size of pre-loaded apps measured by their lines of code (LOC). Although it is impossible for us to get all the source code of the pre-loaded apps, we can still roughly estimate their size based on their decompiled .smali code. Therefore, we can draw a rough estimate of vendor customization from provenance analysis because the number and code size of apps are important indicators. In addition, we also collect firmware release dates and update cycles as supplementary information for our later evaluation (Section 3.3.1).

3.2.2 Permission Usage Analysis

Our next analysis stage is designed to detect instances of permission overprivilege, where an app requests more permissions than it uses. SEFA applies permission usage analysis to measure the adoption of the principle of least privilege in app development. Note that here it is only possible to get the usage of permissions defined in the standard AOSP framework. The usage of vendor-specific permissions cannot be counted because of the lack of related information.

There are four types of permissions in Android: normal, dangerous, system and systemOrSignature. The latter three are sensitive, because normal permissions are not supposed to be privileged enough to cause damage to the user. Specifically, we define permissions declared by element <uses-permission> in the manifest file as requested permissions, and permissions which are actually used (e.g., by using related APIs) as used permissions respectively. An overdeclared permission is a permission which is requested but not used. Overprivileged apps contain at least one overdeclared permission.

Algorithm 1 outlines our process for permission usage analysis. From the database, we have the initial requested permission set of apps (as it is in the manifest information), and our goal is to find the overdeclared permission set. Despite the initial requested permission set, we find it still needs to be

1We use the baksmali [79] tool to translate the Android app’s .dex code into the .smali format.
Algorithm 1: Permission Usage Analysis

Input: initial info. in our database about pre-loaded apps
Output: overdeclared permission set of apps

apps = all apps in one image;
mappings = all permission mappings;
A = API invocation set;
I = intent usage set;
C = content provider usage set;
S = shared user id set;
R = requested-permission set;
U = ∅; // used permission set
O = ∅; // overdeclared permission set

foreach s ∈ S do
    tmp = ∅;
    foreach app ∈ s do
        tmp = tmp ∪ R[app];
    endforeach
    foreach app ∈ s do
        R[app] = tmp;
    endforeach

foreach app ∈ apps do
    foreach a ∈ A[app] do
        U[app].add(mappings[a]);
    endforeach
    foreach i ∈ I[app] do
        U[app].add(mappings[i]);
    endforeach
    foreach c ∈ C[app] do
        U[app].add(mappings[c]);
    endforeach

foreach app ∈ apps do
    O[app] = R[app] − U[app];

return O
augmented. Especially, there is a special manifest file attribute, `sharedUserId`, which causes multiple apps signed by the same developer certificate to share a user identifier, thus sharing their requested permission sets. (As permissions are technically assigned to a user identifier, not an app, all such apps will be granted the union of all the permissions requested by each app. Accordingly, apps with the same `sharedUserId` require extra handling to get the complete requested permission set.) Next, we leverage the known permission mapping built by earlier work [6] to determine which permissions are actually used.\(^2\) Having built both the requested permission set and the used permission set, we can then calculate the overdeclared permission set.

Our approach to calculate the overdeclared permission set is conservative. Notice that some permissions declared in the manifest file may be deprecated in the corresponding standard Android framework. An example is the permission `READ_OWNER_DATA`\(^3\) that was removed after API level 8 (i.e., Android version 2.2), but still declared by one app in the Nexus 4 (API level 17, or Android 4.2). We do not consider them as overdeclared permissions, because the vendor may retain deprecated permissions in the customized framework for its own usage.

After obtaining the overdeclared permission set, we then analyze the overall permission usage of each device, and classify results by provenance. The distributions of overprivileged apps as well as overdeclared permissions can then both be studied. Further, we also perform horizontal and vertical analysis, i.e., cross-vendor, same-generation, vs. cross-generation, same-vendor comparisons.

### 3.2.3 Vulnerability Analysis

While our permission usage analysis aims to measure the software development practices used in the creation of each pre-loaded app, vulnerability analysis is concerned with finding real, actionable exploits within those apps. Specifically, we look for two representative types of vulnerabilities in the Android platform which stem from misunderstanding or misusing Android’s permission system. First, we identify *permission re-delegation attacks* [32], which are a form of the classic confused deputy attack [44]. Such an attack exists if an app can gain access to an Android permission without actually requesting it. A typical example is an app which is able to send Short Message Service (SMS) messages without acquiring the (supposedly required) `SEND_SMS` permission. For the second kind of vulnerability, we consider *content leaks*, which essentially combines the two types of content provider vulnerabilities reported by Zhou and Jiang [101]: *passive content leaks* and *content pollution*. An unprotected content provider (i.e.,

\(^2\)Early studies including PScount [6] concern themselves only with those permissions that are available to third-party apps. In our study, we need to cover additional permissions defined at `system` and `systemOrSignature` levels, which may not be well documented.

\(^3\)Without specification, the prefix of standard Android permission name `android.permission` is omitted in this dissertation.
one that takes no sensitive permission to protect its access) is considered to have a passive content leak if it is world-readable, and to have content pollution if it is world-writable. We extend this definition to cover both open and protected content providers. The protected ones are also interesting as there may also exist unauthorized accesses to them through the other three types of components which could serve as springboards for exploitation. For ease of presentation, we call these vulnerabilities content leaks.

As our main goal is to accurately locate possible vulnerabilities, we in this study consider the following adversary model: a malicious app, which is compatible with the phone, may be installed on the phone by the user. We do not expect the malicious app will request any sensitive permissions during installation, which means it will only rely on vulnerable apps to accomplish its goals: either steal money from the user, gather confidential data, or maliciously destroy data. In other words, we limit the attacker to only unprivileged third-party apps to launch their attacks.

Keeping this adversary model in mind, we focus our analysis on security-critical permissions – the ones that protect the functions that our adversary would most like to gain access to. Specifically, for permission re-delegation attacks, we focus on permissions that are able to perform dangerous actions, such as SEND_SMS and MASTER_CLEAR, because they may lead to serious damage to the user, either financially or in terms of data loss. As for content leaks, we ignore those whose exposures are likely to be intentional. Note that some apps may be vulnerable to low-severity content leaks; for example, publicly-available information about a network TV schedule is not as sensitive as the user’s banking credentials. In other words, we primarily consider serious content leaks whose exposures are likely to cause critical damages to the user.

To actually find these vulnerabilities, we rely on a few key techniques. An essential one is reachability analysis, which is used to determine all feasible paths from the entrypoint set of all Android components, regardless of whether we consider them to be protected by a sensitive permission (Section 3.2.3.1). To better facilitate vulnerability analysis, we define two varieties of sinks:

- **sensitive-sinks**: sensitive Android APIs which are related to sensitive permissions (e.g., MASTER_CLEAR) of our concern
- **bridge-sinks**: invocations that are able to indirectly trigger another (vulnerable) component, e.g., sendBroadcast

Note that any path reachable from an open entrypoint or component can be examined directly to see if it has a sensitive-sink. Meanwhile, we also determine whether it could reach any bridge-link that will trigger other protected components (or paths). The remaining paths, whose entrypoints are protected, are correlated with paths that contain bridge-sinks to form the complete vulnerable path, which is likely

---

4 Some content providers are exported explicitly, such as TelephonyProvider in the app of the same name.
cross-component or even cross different apps. This is essentially a reflection-based attack and we will describe it in Section 3.2.3.2 in greater detail. All calculated (vulnerable) paths will subject to manual verification.

We stress that unlike some previous works (e.g., [40]) which mainly focus on discovery of vulnerabilities, this analysis stage primarily involves a more contextual evaluation of vulnerabilities, including distribution, evolution and the impact of customization. Especially, we use the distribution of vulnerable apps as a metric to assess possible security impact from vendor customizations. Note the detected vulnerabilities are classified into different categories by their provenance and leveraged to understand the corresponding impact of customization. As mentioned earlier, both horizontal and vertical impact analyses are performed.

3.2.3.1 Reachability Analysis

Our reachability analysis is performed in two steps. The first step is intra-procedural reachability analysis, which involves building related call graphs and resolving it by conventional def-use analysis [26]. The resolution starts from the initial state (pre-computed when the database is initially populated) and then gradually seeks a fixed point of state changes with iteration (due to various transfer functions). However, as the state space might be huge (due to combinatorial explosion), the convergence progress could be slow or even unavailable. In practice, we have to impose additional conditional constraints to control the state-changing iteration procedure. We call the result of intra-procedural analysis, i.e., the states of variables and fields, a summary.

The second step is inter-procedural reachability analysis that is used to propagate states between different methods. After each propagation, method summaries might be changed. In such cases, intra-procedural reachability analysis is performed again on each affected method to generate a new summary. Inter-procedural reachability analysis is also an iterative process, but takes longer and requires more space to converge; therefore, we use some heuristics to reduce the computational and space overhead. For instance, if a variable or field we are concerned with has already reached a sink, there is no need to wait for convergence. A more formal description of our reachability analysis is listed in Algorithm 2.

Paths of apps from different vendors but with similar functionality may share something in common, especially for those apps inherited from the standard AOSP framework. Here “common” does not mean that their source code is exactly the same, but is similar from the perspective of structure and functionality. Many devices reuse the code from the AOSP directly, without many modifications. If we have already performed reachability analysis on such a common path, there is no need to do it on its similar counterparts. We believe this improves system performance since reachability analysis is time consuming (especially when the state space is huge). Therefore, we also perform a similarity analysis
Algorithm 2: Reachability Analysis

**Input:** path from entrypoint to sink  
**Output:** path is reachable or not

```plaintext
ret = false;
intra_analysis(all nodes in path);
nodes = nodes in the path;
edges = edges in the path;

while constraint does not meet do
  flag = false;
  foreach n ∈ nodes do
    callee = callee set of n;
    if callee = ∅ then
      break;
    foreach c ∈ callee do
      if (n, c) ∈ edges then
        flag = flag ∪ c.summarize(n);

  if flag then
    inter_analysis(c);
    if constraint meets then
      break;
  else
    break;

ret = reachability_check(path with summary information);
return ret
```

as a part of the reachability analysis to avoid repetitive efforts.

### 3.2.3.2 Reflection Analysis

To facilitate our analysis, we classify vulnerable paths into the following three types:

- **in-component:** a vulnerable path that starts from an unprotected component to a sink that is located in the same component.

- **cross-component:** a vulnerable path that starts from an unprotected component, goes through into other components within the same app, and then reaches a sink.
• **cross-app**: a vulnerable path that starts from an unprotected component of one app, goes through into another app’s components, and eventually reaches a sink.

The in-component vulnerable paths are relatively common and have been the subject of recent studies [40, 55]. However, the latter two, especially the cross-app ones, have not been well studied yet, which is thus the main focus of our reflection analysis. Note that a reflection-based attack typically involves with multiple components that may not reside in the same app. A concrete example that is detected by our tool will be shown in Figure 3.6 (Section 3.3.3.1).

Traditional reachability analysis has been effective in detecting in-component vulnerable paths. However, it is rather limited for other cross-component or cross-app vulnerable paths. (A cross-app execution path will pass through a chain of related apps to ultimately launch an attack.) In order to identify them, a comprehensive analysis of possible “connection” between apps is necessary. To achieve that, our approach identifies not only possible reachable paths within each component, but also the invocation relationship for all components. The invocation relationship is essentially indicated by sending an `intent` [37] from one component to another, explicitly or implicitly. An explicit intent specifies the target component to receive it and is straightforward to handle. An implicit intent, on the other hand, may be sent anonymously without specifying the receiving component, thus requiring extra handling (i.e., *intent resolution* in Android) to determine the best one from the available components. In our system, SEFA essentially mimics the Android intent resolution mechanism by matching an intent against all possible `<intent-filter>` manifest declarations in the installed apps. However, due to the offline nature of our system, we have limited available information about how the framework behaves at run-time. Therefore, we develop the following two heuristics:

- A component from the same app is preferable to components from other apps.
- A component from a different app which shares the same sharedUserId is preferable to components from other apps.

If multiple component candidates still exist for a particular intent, we simply iterate each one to report possible vulnerable path and then manually verify it in a real phone setting. In Algorithm 3, we summarize the overall procedure. The basic idea here is to maintain a visited component list. For a particular component, the algorithm returns ∅ if it has been visited; otherwise we add it into the list, and check all possible components that are able to start up that component recursively.
Algorithm 3: Reflection Analysis

```
Input: current component, visited component list
Output: vulnerable path set

/* find vulnerable paths recursively from backward for current component */
c = current component;
VC = visited component list;
CC = components which are able to start up c;
V = ∅; // vulnerable path set

if c ∈ VC then
    return V
else
    VC.append(c);

foreach cc ∈ CC do
    tmp = VC.clone();
    V.add(Reflection Analysis(cc, tmp));

return V
```

3.3 Implementation and Evaluation

We have implemented a SEFA prototype as a mix of Java code and Python scripts with 11,447 and 4,876 lines of code (LOC) respectively. In our evaluation, we examined ten representative phones (Table 3.1) released between the end of 2010 and the end of 2012 by five popular vendors: Google, Samsung, HTC, LG, and Sony. The selected phone model either has great impact and is representative, or has huge market share. For example, Google’s phones are designed to be reference models for their whole generation; Samsung is the market leader which occupies 39.6% of smartphone market share in 2012 [50]. To analyze these phone images, our system requires on average 70 minutes to process each image (i.e., around 30 seconds per app) and reports about 300 vulnerable paths for us to manually verify. Considering the off-line nature of our tool, we consider this acceptable, even though our tool could be optimized further to speed up our analysis.

3.3.1 Provenance Analysis

As mentioned earlier, the provenance analysis collects a wide variety of information about each device and classifies pre-loaded apps into three different categories. In Table 3.1, we summarize our results. Overall, these ten devices had 1,548 pre-loaded apps, totalling 114,427,232 lines of code (in terms of
decompiled .smali code). A further break-down of these apps show that, among these devices, there are on average 28.2 (18.22%), 99.7 (64.41%), and 26.9 (17.38%) apps from the categories of AOSP, vendor and third-party, respectively. Note that the apps in the AOSP category may also be customized or extended by vendors (Section 3.2.1). As a result, the figures in the AOSP column of Table 3.1 should be considered an upper bound for the proportion of code drawn from the AOSP. Accordingly, on average, vendor customizations account for more than 81.78% of apps (or 76.34% of LOC) on these devices.

In our study, we selected two phone models for each vendor, one from the current crop of Android 4.x phones, and one from the previous generation of 2.x devices. Table 3.2 shows the initial release date of each phone model, as reported by GSM Arena [43]. As it turns out, these devices can be readily classified by their release dates: the past generation of devices all were initially released before 2012, while the current generation’s products were released in 2012 or later. Therefore, we break them down into pre-2012 and post-2012 devices. Such classification is helpful to lead to certain conclusions. For example, as one might expect, the complexity of these devices is clearly increasing over time. In all cases, the post-2012 products from any given manufacturer contain more apps and LOC than their pre-2012 counterparts. Specifically, the HTC Wildfire S has 147 apps with 9,643,448 LOC, and the HTC One X has 280 apps with 19,623,805 LOC. The number of apps and LOC increase 90.47% and 103.49%, respectively.

Table 3.1 Provenance analysis of representative devices

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Device</th>
<th>Version</th>
<th>Build</th>
<th># Apps</th>
<th># LOC</th>
<th>AOSP app</th>
<th>vendor app</th>
<th>third-party app</th>
</tr>
</thead>
<tbody>
<tr>
<td>Samsung</td>
<td>Galaxy S2</td>
<td>2.3.4; I9100XWKI4</td>
<td>172</td>
<td>10,052,891</td>
<td>26 (15.12%)</td>
<td>2,419,155 (24.06%)</td>
<td>114</td>
<td>3,519,955</td>
</tr>
<tr>
<td>Samsung</td>
<td>Galaxy S3</td>
<td>4.0.4; I9300UBALF5</td>
<td>185</td>
<td>17,339,442</td>
<td>30 (16.22%)</td>
<td>6,344,721 (36.59%)</td>
<td>119</td>
<td>5,660,569</td>
</tr>
<tr>
<td>HTC</td>
<td>Wildfire S</td>
<td>2.3.5; CL362953</td>
<td>147</td>
<td>9,643,448</td>
<td>24 (16.33%)</td>
<td>2,759,415 (28.61%)</td>
<td>94</td>
<td>3,514,921</td>
</tr>
<tr>
<td>HTC</td>
<td>One X</td>
<td>4.0.4; CL018532</td>
<td>280</td>
<td>19,623,805</td>
<td>29 (10.36%)</td>
<td>4,718,633 (24.05%)</td>
<td>190</td>
<td>7,354,468</td>
</tr>
<tr>
<td>LG</td>
<td>Optimus P50</td>
<td>2.2; FRG83</td>
<td>100</td>
<td>6,160,168</td>
<td>27 (27.00%)</td>
<td>1,152,885 (18.72%)</td>
<td>40</td>
<td>604,197</td>
</tr>
<tr>
<td>LG</td>
<td>Optimus P60</td>
<td>4.0.4; M388-04</td>
<td>115</td>
<td>12,129,841</td>
<td>28 (24.35%)</td>
<td>3,170,950 (26.14%)</td>
<td>63</td>
<td>7,299,990</td>
</tr>
<tr>
<td>Sony</td>
<td>Xperia Arc S</td>
<td>2.3.4; K19326.A63</td>
<td>147</td>
<td>7,909,131</td>
<td>28 (31.94%)</td>
<td>1,800,690 (23.65%)</td>
<td>122</td>
<td>2,886,971</td>
</tr>
<tr>
<td>Sony</td>
<td>Xperia S</td>
<td>2.3.4; K19326.A64</td>
<td>156</td>
<td>5,234,802</td>
<td>31 (42.47%)</td>
<td>1,109,894 (21.67%)</td>
<td>41</td>
<td>2,821,874</td>
</tr>
<tr>
<td>Google</td>
<td>Nexus S</td>
<td>4.0.4; GRI39HI</td>
<td>71</td>
<td>5,234,892</td>
<td>31 (42.47%)</td>
<td>1,109,894 (21.67%)</td>
<td>41</td>
<td>2,821,874</td>
</tr>
<tr>
<td>Google</td>
<td>Nexus 4</td>
<td>4.2; JOP40C</td>
<td>91</td>
<td>15,848,907</td>
<td>31 (34.07%)</td>
<td>2,506,778 (15.82%)</td>
<td>57</td>
<td>12,156,673</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td></td>
<td>1548</td>
<td>114,427,232</td>
<td>282 (22.86%)</td>
<td>27,072,796 (23.60%)</td>
<td>997</td>
</tr>
</tbody>
</table>

Our analysis also shows that, though the baseline AOSP is indeed getting more complicated over time – but vendor customizations are at least keeping pace with the AOSP. This trend is not difficult to understand, as vendors have every incentive to add more functionality to their newer products, especially in light of their competitors doing the same.

The Google-branded phones are particularly interesting. Both have relatively few apps, as they are

---

5 This category contains apps that exist in the AOSP and may (or may not) be customized by the vendor – Section 3.2.1. This definition also applies to Tables 3.3, 3.4 and 3.5.
### Table 3.2 Release dates of examined Android devices

<table>
<thead>
<tr>
<th>Device</th>
<th>Release Date</th>
<th>Update Date &amp; Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nexus S</td>
<td>Dec 2010</td>
<td>4.0.4, Mar 2012; 4.1.2, Oct 2012</td>
</tr>
<tr>
<td>Nexus 4</td>
<td>Nov 2012</td>
<td>N/A</td>
</tr>
<tr>
<td>Wildfire S</td>
<td>May 2011</td>
<td>N/A</td>
</tr>
<tr>
<td>One X</td>
<td>May 2012</td>
<td>4.1.1, Nov 2012; 4.1.2, Mar 2013</td>
</tr>
<tr>
<td>Optimus P350</td>
<td>Feb 2011</td>
<td>N/A</td>
</tr>
<tr>
<td>Optimus P880</td>
<td>Jun 2012</td>
<td>4.1.2, Mar 2013</td>
</tr>
<tr>
<td>Galaxy S2</td>
<td>Apr 2011</td>
<td>2.3.6, Dec 2011; 4.0.3 Jul 2012; 4.1.1, Jan 2013; 4.1.2, Apr 2013</td>
</tr>
<tr>
<td>Galaxy S3</td>
<td>May 2012</td>
<td>4.1.1, Nov 2012; 4.1.2, Mar 2013</td>
</tr>
<tr>
<td>Xperia Arc S</td>
<td>Sep 2011</td>
<td>4.0.4, Aug 2012</td>
</tr>
<tr>
<td>Xperia SL</td>
<td>Sep 2012</td>
<td>4.1.2, May 2013</td>
</tr>
</tbody>
</table>

1. The release dates may vary by area and carrier, but the difference is not that significant (e.g., around one month).
2. The update dates collected here are mainly for markets in United States.

Designed to be reference designs with only minor alterations to the core AOSP. However, the Nexus 4 has over three times as many lines of code as were included in the Nexus S, despite adding only 18 apps. The Nexus S was the simplest phone we studied, while the Nexus 4 is the third most complex – only the HTC One X and Samsung Galaxy S3, which are well known for their extensive customization, have more LOC. We attribute this to the fact that the Nexus 4 includes newer versions of vendor-specific apps (e.g., Gmail) that have more functionality with larger code size.

Meanwhile, we also observe these devices experience slow update cycles: the average interval between two updates for a phone model is about half a year! Also, it is interesting to note that the updates in the United States tend to lag behind some other areas. For instance, users of the Samsung Galaxy S3 in the United Kingdom received updates to Android 4.1.1 (which was released in July 2012) in September 2012, while the updates were not available for users in the United States until January 2013. If we compare the update dates in Table 3.1 with the corresponding dates of Android releases [38], it often takes half a year for vendors to provide an official update (excepting Google’s reference phones) for users in the United States, though in many cases carriers may share the blame. Overall, official updates can hardly be called timely, which thereby seriously affects the security of these devices.

#### 3.3.2 Permission Usage Analysis

After determining the provenance of each pre-loaded app, our next analysis captures the instances of permission overprivilege, where an app requests more permissions than it uses. The results are summarized in Table 3.3. On average, there is an alarming fact that across the ten devices, 85.78% of apps
are overprivileged. Even Google’s reference devices do not necessarily perform better than the others; the Nexus S has the second most overprivileged apps of all the pre-2012 devices. Note that the situation appears to be getting slightly better as time goes on. The average percentage of overprivileged apps in the post-2012 devices has decreased to 83.61%, compared to 87.96% of all apps on pre-2012 devices. But this situation is still hardly reassuring. Interestingly, our results show that the proportion of overprivileged pre-loaded apps are more than the corresponding result of third-party apps (reported by [6, 33]). We believe there are two main reasons: 1) pre-loaded apps are more privileged than third-party apps, as they can request certain permissions not available to third-party ones; 2) pre-loaded apps are more frequent in specifying the sharedUserId property, thereby gaining many (possibly unnecessary) permissions.

Table 3.3 Permission usage analysis of representative devices

<table>
<thead>
<tr>
<th>Device</th>
<th>% of overprivileged apps among all pre-loaded apps</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
</tr>
<tr>
<td>Nexus S</td>
<td>90.41%</td>
</tr>
<tr>
<td>Wildfire S</td>
<td>92.52%</td>
</tr>
<tr>
<td>Optimus P350</td>
<td>85.00%</td>
</tr>
<tr>
<td>Galaxy S2</td>
<td>88.37%</td>
</tr>
<tr>
<td>Xperia Arc S</td>
<td>83.52%</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>87.96%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Device</th>
<th>% of overprivileged apps among all pre-loaded apps</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
</tr>
<tr>
<td>Nexus 4</td>
<td>79.12%</td>
</tr>
<tr>
<td>One X</td>
<td>78.21%</td>
</tr>
<tr>
<td>Optimus P880</td>
<td>91.30%</td>
</tr>
<tr>
<td>Galaxy S3</td>
<td>87.57%</td>
</tr>
<tr>
<td>Xperia SL</td>
<td>81.82%</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>83.61%</td>
</tr>
</tbody>
</table>

**Overall Average: 85.78%**

Specifically, if we take into account the provenance of each app, a different story emerges. Looking over the breakdowns in Table 3.3, the modest gains over time appear to be primarily attributable to a reduction in app overprivilege among AOSP apps; vendors appear to be responsible for roughly the same amount of overprivileged apps in each image (51.29% of all pre-2012 apps, vs. 52.71% of post-
Figure 3.2: Distributions of overprivileged apps

(a) Pre-2012 devices

(b) Post-2012 devices
2012 apps). In this sense the vendors themselves do not appear to care significantly more about the least privilege principle than third-party app developers do, despite being, on average, much larger corporate entities.

In Figure 3.2, we summarize the distributions of overprivileged apps among pre-2012 devices and post-2012 devices respectively. The majority of overprivileged apps have no more than 10 overdeclared permissions. Note that pre-loaded apps have access to certain permissions that are not available to third-party apps. Therefore, these overdeclared permissions, if exploited, can lead to greater damage. For example, our results demonstrate that both `REBOOT` and `MASTER_CLEAR` are among overdeclared permissions that can allow for changing (important) device status or destroying user data without notification.

### 3.3.3 Vulnerability Analysis

Our vulnerability analysis led to several interesting results, especially once combined with our efforts to determine the provenance of each app. In particular, if we consider the distribution of vulnerable apps across each phone image (Table 3.4), the percentage of vulnerable apps of these devices varies from 1.79% to 14.97%. Applying our horizontal analysis to each generation of devices, it appears that the HTC Wildfire S and LG Optimus P880 have the most vulnerable apps in each generation. Inversely, the Sony Xperia Arc S and – interestingly – the HTC One X have the least vulnerable apps by proportion. As one may expect, Google’s reference phones (especially the Nexus 4) both perform well compared to their contemporaries, as their images are designed to be a reference baseline. Our vertical analysis has an even more impressive result – the percentage of vulnerable apps across each generation dropped from an average of 8.99% last generation to 4.56% this generation. Even incorporating provenance information in the vertical analysis, there is a dramatic improvement for all three categories of app.

However, the percentage of vulnerable apps is not necessarily a good metric to measure the security of such devices. The complexity metrics we collected as part of our provenance analysis (see Table 3.1) show that the devices contain ever-increasing amounts of code. Therefore, while fewer vulnerabilities – as a percentage – may be introduced in newer phones, the sheer scale of their stock images works to counteract any gains. Furthermore, some vulnerable apps may contain more vulnerabilities than others. As a result, there is value in counting the absolute number of critical vulnerabilities as well as the proportion of vulnerable apps.

To this end, we summarize our results in that format in Table 3.5. When we use this table for vertical differential analysis, it tells quite a different story. While, indeed, the number of vulnerabilities in each generation of devices generally decreased, the reduction is nowhere near so dramatic. Furthermore, when considered in horizontal differential analysis, different devices take the crown for most and least secure devices in each generation. The HTC Wildfire S is still the least secure pre-2012 device, but only
Table 3.4 Vulnerability analysis of representative devices (I)

<table>
<thead>
<tr>
<th>Device</th>
<th>Pre-2012 devices</th>
<th>% of vulnerable apps among all apps</th>
<th>Post-2012 devices</th>
<th>% of vulnerable apps among all apps</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Total</td>
<td>AOSP app</td>
<td>vendor app</td>
</tr>
<tr>
<td>Nexus S</td>
<td>5.48%</td>
<td>2.74%</td>
<td>2.74%</td>
<td>0.00%</td>
</tr>
<tr>
<td>Wildfire S</td>
<td>14.97%</td>
<td>4.76%</td>
<td>8.84%</td>
<td>1.36%</td>
</tr>
<tr>
<td>Optimus P350</td>
<td>11.00%</td>
<td>4.00%</td>
<td>1.00%</td>
<td>6.00%</td>
</tr>
<tr>
<td>Galaxy S2</td>
<td>12.21%</td>
<td>3.49%</td>
<td>6.98%</td>
<td>1.74%</td>
</tr>
<tr>
<td>Xperia Arc S</td>
<td>2.27%</td>
<td>1.14%</td>
<td>0.00%</td>
<td>1.14%</td>
</tr>
<tr>
<td>Average</td>
<td>8.99%</td>
<td>3.23%</td>
<td>3.91%</td>
<td>1.85%</td>
</tr>
</tbody>
</table>

Overall Average: 6.77%

by a hair – the Samsung Galaxy S2 has only one fewer vulnerability. The Sony Xperia Arc S is tied with the Google Nexus S for the most secure pre-2012 device. Meanwhile, there is a complete shake-up among the post-2012 devices: the Samsung Galaxy S3 has 40 vulnerabilities to the LG Optimus P880’s 26, while the HTC One X (at 15 vulnerabilities) falls to mid-pack, behind the Nexus 4 (at 3) and the Sony Xperia SL (at 8).

Table 3.5 still does not tell the complete story. Looking at the table’s provenance analysis results, it appears that most of the vulnerabilities stem from the AOSP. However, recall that our provenance analysis concerns the original provenance of each app, not each vulnerability. To gather information about the provenance of each vulnerability, we manually examine each of the reported vulnerable paths. Our results are shown in Figure 3.3 (note that Google’s phones are not included here because the AOSP is led by Google, making the distinction difficult). For the Samsung, HTC, and LG phones, the majority of vulnerabilities – between 64.71% and 85.00% – did not originate from the AOSP. However, for both of Sony’s products, only 37.50% of vulnerabilities were caused by vendor customizations. In fact, one of Sony’s modifications to the AOSP actually mitigated a pre-existing bug in it.

We can also apply vertical differential analysis to this data, and therefore look at the evolution
of vulnerabilities over time. The post-2012 devices may have inherited some vulnerabilities that were never caught during the lifetime of the pre-2012 devices, as they often have code in common with earlier devices by the same manufacturer\(^6\); alternatively, they may have introduced new vulnerabilities in their new and altered features. Figure 3.4 depicts this evolutionary information, which varies wildly in proportion between different vendors. For example, for the HTC One X, about 60.00\% of its vulnerabilities were inherited from the previous device, while the Samsung Galaxy S3 has more introduced vulnerabilities than inherited ones (47.50\% vs. 35.00\%).

No treatment of this topic would be complete without discussing the distribution of vulnerabilities that we found. Table 3.6 lists the vulnerabilities we focus on. We use the names of permission to represent the most common (i.e., shared by devices of different vendors) vulnerabilities for permission re-delegation attacks with explicit permission names, and use OTHER to represent all other vulnerabilities (including both types of studied vulnerabilities). Note that vulnerabilities belong to OTHER do not mean they are not critical, and the only reason is that they are more vendor- and model-specific. Table 3.7 lists the distribution of these vulnerabilities.

With these detected vulnerabilities, we have attempted to contact the corresponding vendors. While some of them have already been confirmed, other vendors have still not spoken with us even after several months. Among the detected vulnerabilities, we believe cross-app ones are the most challenging to

\(^6\) The relationship between each pair of devices may be not direct (i.e., predecessor and successor), but we can still regard these vulnerabilities as an inheritance because they are vendor-specific.
Table 3.5 Vulnerability analysis of representative devices (II)

<table>
<thead>
<tr>
<th>Pre-2012 devices</th>
<th># of vulnerabilities</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>AOSP app</td>
<td>vendor app</td>
<td>third-party app</td>
</tr>
<tr>
<td>Nexus S</td>
<td>8</td>
<td>6</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Wildfire S</td>
<td>40</td>
<td>23</td>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>Optimus P350</td>
<td>17</td>
<td>11</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Galaxy S2</td>
<td>39</td>
<td>18</td>
<td>18</td>
<td>3</td>
</tr>
<tr>
<td>Xperia Arc S</td>
<td>8</td>
<td>6</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Post-2012 devices</th>
<th># of vulnerabilities</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>AOSP app</td>
<td>vendor app</td>
<td>third-party app</td>
</tr>
<tr>
<td>Nexus 4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>One X</td>
<td>15</td>
<td>12</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Optimus P880</td>
<td>26</td>
<td>17</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>Galaxy S3</td>
<td>40</td>
<td>20</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>Xperia SL</td>
<td>8</td>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

1 This category contains apps that exist in the AOSP and may (or may not) be customized by the vendor – Section 3.2.1.

detect. Figure 3.5 gives the percentage of cross-app vulnerabilities. On average, 8.90% vulnerabilities are of this type. In the following, we describe some specific vulnerabilities introduced due to vendor customizations.

3.3.3.1 Case Study: Samsung Galaxy S3

In the Samsung Galaxy S3, there is a pre-loaded app named Keystring_misc. This particular app has a protected component PhoneUtilReceiver that leads to a dangerous path for performing a factory reset, thus erasing all user data on the device. This path ends in the phoneReset method, which will broadcast an intent android.intent.action.MASTER_CLEAR to perform the operation. At first sight, this path seems to be safe because this component is protected by the com.sec.android.app.phoneutil.-permission.KEYSTRING permission, which is defined with the restrictive systemOrSignature protection level (i.e., only other firmware apps, or apps from Samsung, can invoke it).

Unfortunately, there exists another app named FactoryTest that contains a feasible path which is able to start up this very component in the Keystring_misc app. This arrangement is an example of a cross-app vulnerable path, which can be used to launch a reflection attack (Figure 3.6). Specifically, this app exports a service called FtClient without any protection. After being launched, the
service will start a thread and then try to build connections with two hard-coded local socket addresses: FactoryClientRecv and FactoryClientSend. The established connections can be exploited to send commands through the first socket. Our manual investigation shows that there are many dangerous operations that can be triggered by this exploit, including MASTER_CLEAR, REBOOT, SHUTDOWN and SEND_SMS.

In our study, we also discover a number of other vulnerabilities. For example, there are four content providers in the sCloudBackupProvider app (with the package name of com.sec.android.sCloud BackupProvider). They expose access interfaces to specific databases, including calllogs.db, sms.db, mms.db and settings.db. Each of them is protected by two permissions, but with normal (non-sensitive) permissions. Commonly, these permissions are used in everyday operations or will not be regarded as very dangerous.

**Table 3.6 Classification of detected vulnerabilities**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL_PRIVILEGED</td>
<td>Initiate a phone call (including emergency number) without requiring confirmation</td>
</tr>
<tr>
<td>MASTER_CLEAR</td>
<td>Wipe out user data and factory reset</td>
</tr>
<tr>
<td>REBOOT</td>
<td>Reboot the device</td>
</tr>
<tr>
<td>RECORD_AUDIO</td>
<td>Allows an application to record audio.</td>
</tr>
<tr>
<td>SEND_SMS</td>
<td>Allows an application to send SMS messages.</td>
</tr>
<tr>
<td>SHUTDOWN</td>
<td>Power off the device</td>
</tr>
<tr>
<td>WRITE_SMS</td>
<td>Allows an application to write SMS messages.</td>
</tr>
<tr>
<td>OTHER</td>
<td>All the other dangerous/critical operations</td>
</tr>
</tbody>
</table>
Table 3.7 Distribution of vulnerabilities among examined devices

<table>
<thead>
<tr>
<th>Name</th>
<th>Google</th>
<th>HTC</th>
<th>LG</th>
<th>Samsung</th>
<th>Sony</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL_PRIVILEGED</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>MASTER_CLEAR</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>REBOOT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>RECORD_AUDIO</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>SEND_SMS</td>
<td>3</td>
<td>6</td>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>SHUTDOWN</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>WRITE_SMS</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>OTHER</td>
<td>2</td>
<td>1</td>
<td>25</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>Total</td>
<td>8</td>
<td>3</td>
<td>40</td>
<td>15</td>
<td>17</td>
</tr>
</tbody>
</table>

Figure 3.5 Distribution of cross-app vulnerabilities

protection levels. Apparently, they are accessible to any third-party app. Also, notice that this app exports four receivers without any protection, and it is able to craft specific intents that trigger corresponding backup actions to save various private information into these databases (e.g., standard SMS messages stored in `mmssms.db` will be copied into `sms.db`). After that, any app can simply retrieve these sensitive information (e.g., SMS messages and contacts information) through the corresponding four content providers without acquiring any sensitive permission.

3.3.3.2 Case Study: LG Optimus P880

The LG Optimus P880 has a number of vulnerabilities; here, we will detail two of them, leading to a permission re-delegation attack and a content leak respectively. However, unlike the Samsung vulnerabilities, neither of the ones we will describe are reflection attacks, making them easier to detect,
Figure 3.6 An example path of reflection attack
describe, and exploit.

The first one is related to REBOOT, a permission reserved for system or pre-loaded apps. In the LG Optimus P880, there is an app named LGSettings, which is a customized version of the AOSP Settings app. This particular app exports an activity com.android.settings.Reboot (completely without permission protection). This activity will be triggered if an android.intent.action.MAIN intent is received, and its invocation will simply reboot the device directly. Note that the AOSP does not have the corresponding vulnerable component.

The second one is a rather direct content leak vulnerability. The com.lge.providers.lgemail content provider in the LGEmail app is not protected, and therefore exposes access to the EMAIL.db, a database that contains three tables named EAccount, EMessageBox and EMessage. These tables are very sensitive, as through them, all account and message related information (including message contents) can be exposed. Note that this app is customized from the AOSP Email app; however, in the AOSP, the corresponding content provider is protected by a permission named com.android.email.permission.ACCESS_PROVIDER with the systemOrSignature protection level. Therefore, LG’s customization here adds a vulnerability to an otherwise-secure AOSP app.

3.4 Discussion

While collecting the data for our evaluation, we saw some indirect evidence of software development policies in place at the various vendors. This evidence may be anecdotal, but we feel it is interesting enough to warrant mention. For example, Sony’s standout performance does not appear to be accidental; in both their devices that we studied, the eventstream content provider (which was implemented as an SQLite database, as many are) actually had explicit checks for SQL injection attacks. Furthermore, Sony’s customized version of the AOSP Mms app actually mitigated problems found in the unaltered problem. Similarly, as we remarked in Section 3.2.3, HTC made considerable progress between the release of the HTC Wildfire S and the One X, possibly due to early exposure of a large proportion of security vulnerabilities in earlier HTC’s devices [73] and the efforts made by the corporation to take security to heart ever since. The One X makes extensive use of custom permissions to prevent further vulnerabilities from creeping into its firmware – a relatively straightforward approach to take, yet an effective one, as shown by our vulnerability analysis.

We also note that there are not very strong correlations between a number of superficial metrics and the number of vulnerabilities present in a stock firmware image. Code size does not strongly correlate: the Nexus 4 has the third-largest number of LOC, but the lowest number of vulnerabilities in the whole study. The number of apps does not correlate: both Sony devices perform very well, despite having a very large number of apps, while the LG devices do poorly on security even though they have the fewest
apps of any non-reference device. Finally, even popularity does not appear to correlate with security: Samsung’s Galaxy S3 was the most popular smartphone of 2012, having shipped in excess of 50 million units as of March 2013 [87], and yet it had the most vulnerabilities of any phone in its generation studied in this chapter.

Lastly, we would like to acknowledge some of the limitations of our work. We do not cover the customization of system level code, which can be an interesting topic for future research. Our current prototype also has several constraints. First of all, our static analysis produces a relatively high false positive rate. On average, our analysis produces less than 300 paths per device. While it does not make too much effort to manually verify each path, it would be better if we could use a light-weight dynamic analyzer to reduce the manual workload. Secondly, we generate the call graph recursively. In order to avoid very deep (potentially infinite) recursion, we constrain our analysis in two ways: only acyclic graphs are allowed, and/or the maximum length of a path (or the maximum exploration time) is set as an upper boundary for safety. These constraints may prevent us from discovering certain vulnerabilities if there exists heavy code obfuscation that either extends the length of vulnerable path or modifies the sinks we use to define such paths. Fortunately, it is our experience that most pre-loaded apps (other than bundled third-party apps) are not extensively obfuscated. As a result, these constraints were primarily triggered by infinite recursion, not by overly long, yet valid, call chains.

3.5 Summary

In this chapter, we quantitatively evaluate the security impact of app-level vulnerabilities in Android devices by designing and implementing the SEFA analysis framework. This tool performs several analyses to study the provenance, permission usage and vulnerability distribution of the pre-loaded apps that make up a device’s firmware image. We evaluated ten devices from five different vendors: two models from each vendor, representing two different generations. We then compare the various vendors’ offerings for a given generation, as well as the evolution of any given vendor’s security practices over time. Our results show that all examined devices are vulnerable. Specifically, due to heavy vendor customizations, on average, over half of the apps in each image are overprivileged vendor apps, and more than 60% of the vulnerabilities we identified stemmed from the vendors’ modifications to the firmware. Furthermore, for most of the manufacturers in our study, these patterns were stable over time, highlighting the need for heightened focus on security by the smartphone industry.
Chapter 4

Detection of Kernel-Level Vulnerabilities

In Chapter 3 we discussed the detection of app-level vulnerabilities. In this chapter, we address kernel-level vulnerabilities.

4.1 Introduction

Android relies on security features provided by Linux kernel [35], including the uid-based permissions model, process level isolation and a built-in mechanism for secure IPC. At its root, Linux kernel provides the fundamental trust base for Android. On top of that, additional defense mechanisms, e.g., user space ASLR, have been adopted by the AOSP in recent years.

However, Linux kernel is not as solid as one might believe. Although Linux has been accepted widely and is trusted for commercial deployment [35], one recent study [95] demonstrated that, among all the other products evaluated in the study, Linux kernel was one of the platforms with the greatest number of vulnerabilities, as published by Common Vulnerabilities and Exposures (CVE). Intuitively, kernel-level vulnerabilities can be used to bypass defense mechanisms that rely on Linux kernel as their trust base. In reality, it has been observed and shown for decades [95] that kernel-level vulnerabilities have been used to perform privilege escalation attacks on desktop PCs.

Unfortunately, the impact of kernel-level vulnerabilities on different Android devices has not been investigated systematically. Although privilege escalation attacks on Android platform have been discussed for years, and vulnerabilities that can be used to gain root privilege have already been discovered and exploited in the wild [59], few studies have focused on the impact of kernel-level vulnerabilities in commodity Android devices. In particular, most of the work has focused primarily on the presence of the vulnerabilities, without verifying further whether or not they were truly exploitable. Similarly, we

\[1\] Kernel vulnerability and kernel-level vulnerability are interchangeable in this chapter.
have little knowledge about the effectiveness of the defense mechanisms deployed. Chipset manufacturers and device vendors always advertise the defense mechanisms they deployed; however, in practice, their effectiveness has not been quantified.

In this chapter, we aim to study kernel-level security issues in commodity Android devices. Specifically, we investigate the impact of kernel-level vulnerabilities in order to, 1) reveal the effectiveness of the kernel defense mechanisms deployed, and 2) help discover new vulnerabilities. The defense mechanisms in this chapter refer to those that may help defend or mitigate kernel-level exploits, i.e., blocking the execution of the corresponding exploits (discussed in Section 4.2). Note that our goal is to evaluate the effectiveness of defense mechanisms deployed by a device as a whole, rather than that of a particular defense mechanism. In addition, we measure the update time of community and device vendors to study the possible open time window of vulnerabilities to attackers.

To facilitate our analyses, we propose a framework referred to as KSEFA (Kernel-level Security Evaluation Framework for Android) to evaluate devices with target images. Given a particular phone image, KSEFA first extracts necessary information from the image, and then verifies and detects possible vulnerabilities. On the one hand, KSEFA is capable of measuring the effectiveness of defense mechanisms quantitatively by verifying whether or not the known vulnerabilities can actually be exploited to perform privilege escalation. Specifically, we selected eight known kernel-level vulnerabilities, and nine typical devices from three representative vendors (Google, Samsung and HTC). We chose three phone models from each vendor, and collected their updated firmware images to perform the evaluation. Our empirical results show that the known kernel-level vulnerabilities can affect the security of Android devices significantly. Firstly, based on our results, newly publicly disclosed kernel-level vulnerabilities could be leveraged to compromise 84.78% devices (with the latest images). Secondly, even after three months, 47.83% of devices with the latest updated images were still vulnerable to the aforementioned vulnerabilities. Lastly, our results also demonstrate that the defense mechanisms deployed in Android devices are not as effective as expected. On the other hand, based on the insights obtained, we are able to build a KSEFA to discover certain zero-day vulnerabilities by performing a static disassembly-level data-flow analysis. In fact, several zero-day vulnerabilities have been discovered using KSEFA, and these findings demonstrate the effectiveness and promise of the system proposed. In addition, we evaluate vendors’ response times to release the official patches. The results show that the responses can hardly be regarded as timely (230 days on average), which leaves a wide open time window for exploitations.

The rest of this chapter is organized as follows. We present our methodology and system framework in Section 4.2, and describe the implementation and evaluation of our results with case studies in Section 4.3. Finally, we discuss and summarize our findings in Sections 4.4 and 4.5, respectively.
4.2 Design

Our goal here is to study the impact of kernel-level vulnerabilities and reveal the effectiveness of the defense mechanisms deployed. To this end, we first collect representative vulnerabilities and devices with target firmware images, and then prepare corresponding exploits and verify whether or not those exploits can be used to perform privilege escalation. The results are then used to demonstrate the effectiveness of the defense mechanisms deployed. Further, based on the insights obtained from the analysis of known vulnerabilities, we are able to discover certain zero-day vulnerabilities by performing a static disassembly-level data-flow analysis. Finally, we study the response time of each participant in the ecosystem quantitatively, including the Linux community, chipset manufactures and device vendors, to measure the security of the Android devices from another perspective.

We collect vulnerabilities based on the time they appeared, their type and provenance. The vulnerabilities collected are distributed among the time periods we focus on, and they have to be sufficiently representative to cover different types of vulnerabilities (e.g., stack overflow and heap overflow). The provenance is also of concern. Kernel-level vulnerabilities might be introduced by either the Linux kernel or device drivers provided by the chipset manufacturers. We call them kernel-generic and manufacturer-specific, respectively. Kernel-generic vulnerabilities are more likely to be generic, but sometimes require more effort in order to make the corresponding exploits work properly. Different kernel images also may be built with different configurations, and some may introduce extra constraints for exploitation. Manufacturer-specific vulnerabilities, on the other hand, do not necessarily lead to limited risks. For example, if one particular chipset is adopted widely by a large number of devices, the corresponding vulnerabilities have significant effects.

We also collect the typical device models according to the same principle to collect vulnerabilities. However, the criterion used to collect firmware images is different. To make the evaluation more convincing, the collected firmware images we investigated have to be the latest official updates at some specified time. For each specific vulnerability, we collect images based on two observation points, i.e., the publicly known date of the vulnerability and a date some time after (e.g., several months) the publicly known date. The former is used to describe the status on the first date, while the latter must be set appropriately, so that the device vendors would have enough time to act on.

Figure 4.1 depicts the architecture of the proposed KSEFA system. Our system takes a device with a target image as its input, preprocesses the image, and imports information extracted into a database for in-depth analyses. Specifically, two analyses are conducted to verify known vulnerabilities and discover unknown ones, respectively. Note that initially the vulnerability set (the far right rectangle in Figure 4.1)
contains only those collected; new discoveries would be added to it once confirmed. In the following, we first describe the preprocessing step in Section 4.2.1, and then introduce the methodology, corresponding challenges, and the solutions of the two analyses in Sections 4.2.2 and 4.2.3, respectively.

### 4.2.1 Preprocessing Firmware Images

In this step, we collect the information necessary for further analyses. As one might expect, information from each device is necessary and important to a feasible exploit. It is known that the offsets of certain fields in some kernel data structures may vary in some cases (e.g., devices with the same kernel, but different configurations), and they must be located precisely for exploitations. In addition, our further analysis also requires information to determine whether some defense mechanisms exist already. Therefore, to serve our needs, the system proposed must be able to extract and analyze information automatically as much as possible. We assume that we already have root privilege to access all related information. This assumption is reasonable, because attackers can always obtain such information from other rooted devices with the same firmware image. In reality, it is possible in most cases to obtain root privilege by flashing a custom recovery image (e.g., CWM [24]); otherwise, we have to use other methods that are discussed later.

We collect and analyze both static and dynamic information. The former (e.g., syscall table address) is fixed and can be extracted directly from an image; the latter (e.g., values of some fields of a given process’ task_struct) is runtime information, which has to be obtained from a device flashed with that particular firmware image. In addition, due to the different configurations (or constraints) enforced by different images, different approaches have to be explored and adopted to achieve our goals.

Most information can be collected simply, i.e., by leveraging some available kernel options. For example, we are able to obtain kernel symbols (/proc/kallsyms) and debug information (/proc/kmsg)
by enabling kptr_restrict and dmesg_restrict, respectively. In fact, although static information (e.g., syscall table address) can always be obtained, dynamic information, such as content in the page table, cannot be extracted easily. Usually we can access the kernel memory to perform the fetch by reading a special device file named /dev/kmem. Although CONFIG_DEVKMEM has been disabled by default since kernel 2.6.x and has been phased out in desktop distributions for a long time, most Android devices still allow this option to be enabled, so that we can access /dev/kmem. Alternatively, we are able to obtain the information required by using the Loadable Kernel Module (LKM), which is also supported by most Android devices. In our experience, in most cases, at least one of the two approaches is feasible to meet our demands. Google’s Nexus 4 with Android 4.4.4 (released on June 2014) is such an example. As shown in Figure 4.2, LKM is disabled, while /dev/kmem remains.

If neither of the two approaches was available for a given device (or even worse, we were not able to obtain root privilege by flashing a custom recovery image), we try to exploit existing vulnerabilities that are capable of performing kernel information leakage. For example, CVE-2013-6282 [22] allows us to read kernel data arbitrarily through the get_user interface.

In summary, the combination of the aforementioned approaches is able to provide us with the information necessary, and most of the processing can be performed automatically.

### 4.2.2 Verifying Known Vulnerabilities

The goal of this analysis is to verify whether or not the known vulnerabilities can actually be exploited to perform privilege escalation. For a given vulnerability, we first determine whether a device with the target firmware image has been patched or not. This can be accomplished by examining the correspond-

---

4 Both are in folder /proc/sys/kernel/.
5 We can also read /dev/mem, which is the same as /dev/kmem, except that physical memory rather than the kernel virtual memory is accessed [56].
ing boot partition. If the device has not been patched, we try to prepare and execute feasible exploits to gain root privilege. Note that the exploitation asks for an executable environment; however, unlike the preprocessing step, we do not expect that the execution is capable of requesting any privileged permissions.

Here, we provide some definitions to make our descriptions clear in what follows. Although the criteria for a successful exploit may vary in different scenarios, we argue that (almost) all of them seek stable exploits. Stable indicates that the particular exploit is not likely to lead to a system crash; otherwise such an exploit cannot be regarded as deterministic, and even malicious use would be encumbered. Therefore, in our research, we define a feasible exploit as one that is also stable.

The complete processing of an exploit in our study can be divided into three stages: triggering, escalating and launching payload. A feasible exploit must be able to trigger the corresponding vulnerability at the very beginning. After that, it must be able to perform escalation to obtain root privilege (by modifying related credentials). Finally, an exploit always emits its payload to affect some system behavior. In many cases, if not most, it leads to the installation of some SU-like (Super User) binaries. 6

<table>
<thead>
<tr>
<th>Data</th>
<th>kernel text (code)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>control-data</strong></td>
<td>syscall table</td>
</tr>
<tr>
<td></td>
<td>return address of kernel stack</td>
</tr>
<tr>
<td></td>
<td>function pointers of kernel data structure</td>
</tr>
<tr>
<td><strong>non-control-data</strong></td>
<td>related fields of kernel data structure</td>
</tr>
</tbody>
</table>

The way to trigger a vulnerability depends highly on the type of that vulnerability. Some vulnerabilities can be triggered relatively easily. For example, CVE-2013-4738 [20], a stack overflow vulnerability that originates in Qualcomm’s device driver. This can be triggered by feeding the corresponding device file a crafted buffer of the proper size. However, some other vulnerabilities (e.g., CVE-2014-3153 [23]) may require very subtle triggering techniques that rely heavily on experience.

A successful escalation achieves two goals. Firstly, it bypasses the defense mechanisms deployed. For example, it is impossible to perform an escalation by modifying kernel text (code) which is read-only enabled. Secondly, it recovers from any inconsistent state that might lead to a system crash. This work is necessary to make the exploit stable, and is particularly important for some complicated escalations–

6Of course, it does not necessarily require such an installation. Nonetheless, certain activities must be performed; otherwise, obtaining root access is meaningless.

41
e.g., escalation by modifying kernel data, which may have something to do with certain kernel resources (e.g., mutex locks), or by affecting the manipulation of corresponding kernel data structure (e.g., linked list). In addition, to help prepare feasible exploits, we classify the escalating targets into two categories: text and data. According to [75], the latter can be categorized further as control-data and non-control-data. Control-data refers to data that would be loaded to the processor program counter (i.e., the PC register in ARM). We summarize the possible escalating targets in Table 4.1.

Payload is relatively independent. At this stage, one must have gained the root privilege. Theoretically, process with root privilege can do whatever it likes, but in practice, it is affected by defense mechanisms that are able to detect and maintain code or file system integrity. In such a case, some behaviors, such as installing SU–like binaries, are not likely to be feasible. The payloads can be divided into two categories: permanent and temporal. The former may break the data integrity of some subsystems (e.g., the file system).

We have to analyze all the vulnerabilities collected manually. For those whose detailed Proofs Of Concept (POCs) have already been provided in the wild (e.g., the contributions of [1] are very valuable to our study), our burden is largely alleviated. Otherwise, for vulnerabilities with few (or even no) publicly available references, we have to study and implement everything by ourselves. In these cases, we try to understand the essence of the vulnerabilities, and thereby prepare feasible exploits to address them. Exploit preparation inevitably requires considerable human participation; however, some efforts can be made to simplify the tasks. Recalling the three stages mentioned above, we adopt a modular design to make code reusable as much as possible.

Sometimes it is necessary to attempt different approaches to exploit a given vulnerability, because the presence of defense mechanisms may block some of the exploits prepared. For example, PXN (Privileged Execute-Never) prevents the execution of arbitrary user space code from the kernel space. As a result, some well-known approaches, e.g., executing user space shellcode by modifying the syscall table, are no longer effective. However, this does not necessarily mean that the corresponding vulnerability is not exploitable, because we can bypass PXN by manipulating related kernel data rather than changing the execution flow. We have made efforts to explore exploitation approaches in order to achieve the most complete coverage possible. However, this is not trivial due to the complicated customizations made by the device vendors. For instance, the size or layout of some kernel data structures may be modified by vendors to support special needs. Theoretically, this problem can be solved by enumerating all of the data structures of the kernel. However, it is an unnecessary waste, because in most cases, that particular data structure remains the same. In such a case, manual investigation may be a better choice. Thus, we may have to bear some necessary trade-off when examining the failed exploits.

However, it is very burdensome to explore all possible approaches to preparing exploits, and in fact, we can avoid many unfeasible trials by investigating the information obtained from the target images.
(see Section 4.2.1), as this may reveal the presence of effective defense mechanisms. If this does not work, we may have to implement and execute the corresponding verifying program to perform the investigation by observing the execution results. For example, escalation with shellcode jumping to the user space can be used to determine the existence of PXN. In addition, in some cases we have to audit the debug information of the verifying program manually. The audit is not straightforward, particularly if none of the debug information (e.g., warning or panic) can be observed from the device (e.g., neither /proc/kmsg nor /proc/last_kmsg is available). In such cases, we must use indirect approaches (e.g., building a new kernel image with the necessary debug information if the source code is provided by the vendor). Fortunately, most of the target images are generous in providing the information required.

In order to evaluate the integral effectiveness of the defense mechanisms a device deploys, we transfer it into a flow network problem. For a given vulnerability, \( \text{vul}_i \), we define full exploiting approach sets, \( T_i \), \( E_i \) and \( L_i \), for the three stages, respectively. Note that the full sets include exploiting approaches that are feasible in a hypothetical device without deploying any defense mechanism, and different vulnerabilities may have different full exploiting approach sets. The cardinality of each set represents the number of exploiting approaches. We also define two pseudo nodes \( s \) and \( d \), which refer to source and destination, respectively. A path is defined as a sequence of nodes from node \( s \) to node \( d \). For the given vulnerability, \( \text{vul}_i \), the total number of paths is calculated as:

\[
N_i = |T_i| \cdot |E_i| \cdot |L_i| \quad (4.1)
\]

This equation suggests that each vertex in a previous stage can reach every vertex in the following stage. This is true in our study because of the essence of the exploitation. As an example, in the triggering stage, no matter how many possible approaches there are, all approaches must share the same property (i.e., in user space or kernel space) after the successful triggering, as they will thereby share the same approaches in the escalating stage.

For a given vulnerability, \( \text{vul}_i \), and a device, \( \text{dev}_j \), the exploitation environment can be modelled as a flow network, i.e., a Directed Acyclic Graph (DAG) \( G^{ij} = (V^{ij}, E^{ij}) \), where \( V^{ij} = \{v^{ij}_1, v^{ij}_2, \ldots \} \) is the set of \( \{s, d\} \cup T_i \cup E_i \cup L_i \), and \( E^{ij} = \{(v^{ij}_p, v^{ij}_q) | v^{ij}_p, v^{ij}_q \in V \} \). The edge, \( E^{ij} \), describes the transition between stages, which might be blocked by some defense mechanism deployed in device \( \text{dev}_j \). \( G^{ij} \) is connected if, and only if, there is at least one unblocked path. As shown in Figure 4.3, the solid line with an arrow indicates a transition between stages that is feasible, and the dashed line with an arrow indicates that it is blocked. For \( \text{vul}_i \) in device \( \text{dev}_j \), we use \( T^{ij}_i \), \( E^{ij}_i \) and \( L^{ij}_i \) to represent the sets of feasible paths for the three stages, respectively, as shown in Equation 4.2.

\[
N^{ij}_i = |T^{ij}_i| \cdot |E^{ij}_i| \cdot |L^{ij}_i| \quad (4.2)
\]
Therefore, for the given vulnerability $vul_i$, the effectiveness of the defense mechanisms deployed in device $dev_j$ can be calculated with Equation 4.3.

$$e_j^i = 1 - \frac{N^j_i}{N_i}$$  \hspace{1cm} (4.3)  

If no path is blocked (i.e., $e_j^i = 0$), then the effectiveness is defined as useless to vulnerability $vul_i$; otherwise, it is said to be effective. Accordingly, the calculated value of the effectiveness in the example shown in Figure 4.3 is 0.33. In particular, if there exists at least one cut that is able to block all edges between any two consecutive stages (i.e., $e_j^i = 1$), we say that the effectiveness of the defense mechanisms deployed in device $dev_j$ is complete to vulnerability $vul_i$. Note that this model has its limitations, because it is unsuitable in evaluating the defense mechanisms that are used to mitigate kernel information leakage; however, because such mechanisms can always be bypassed (as discussed in Section 4.2.1), we do not evaluate them in our study.

### 4.2.3 Discovering Unknown Vulnerabilities

Based on the insights obtained from the analysis of known vulnerabilities in Section 4.2.2, i.e., some useful patterns of the known vulnerabilities, we are able to discover unknown (new) manufacturer-specific vulnerabilities by conducting a static disassembly-level data-flow analysis.

Intuitively, manufacturer-specific vulnerabilities are often related to operations that manipulate certain device files through system calls, either `ioctl` or `mmap`, both of which might be exploitable if user input is allowed without verifying the address or size. For example, if the `ioctl` interface allows the user
to overwrite arbitrary addresses with arbitrary values\textsuperscript{7}, it is easy to gain the root privilege by overwriting the critical kernel addresses (e.g., credentials in \texttt{task\_struct}) with prepared values. We summarize the observed patterns of such vulnerabilities in Table 4.2. These patterns are the basis of our proposed system. Similar to the work of SEFA, the problem is transferred into a task to find vulnerable paths from the source to the sink again, i.e., whether or not any path exists that allows the manipulable user input to modify the content of kernel memory addresses.

\begin{table}[h]
\centering
\caption{Patterns of manufacturer-specific vulnerabilities}
\begin{tabular}{|c|c|}
\hline
Source & Sink \\
\hline
\texttt{ioctl/write} & Overwriting specified kernel address \\
& Overwriting kernel stack address with specified size \\
\texttt{ioctl} & Invoking \_\_copy\_from\_user \\
\texttt{mmap} & Invoking \texttt{remap\_PFN\_range} \\
\hline
\end{tabular}
\end{table}

Recall that we used the basic data-flow analysis in SEFA (i.e., Section 3.2.3.1), which performs a reachability analysis at the IR language level rather than the source code level (i.e., Jimple and Java, respectively, but the former is very similar to the latter). Therefore, we try to conduct a similar analysis in KSEFA. However, this becomes complicated, as the target object of analysis becomes the kernel image, and it is not easy to recover the semantic meaning from the disassembly code. Specifically, we must deal with registers and instructions, rather than variables and statements. To this end, we record registers (and memory addresses as well) as the variables they represent at different places, in order to maintain the semantics of the data-flow on which we focus. Apparently, in this case, those memory addresses are the key objects in the corresponding state transition matrices of the reachability analysis. Note that sometimes the target memory address cannot be extracted from the instruction directly, and we have to create an artificial one. For example, we create two memory addresses, \texttt{mem[r4 + 4] and mem[r4 + 8]}, for instruction \texttt{ldmib r4, \{r1, r2\}}.

Figure 4.4 shows the flow to discover unknown vulnerabilities. The kernel image is first fed into a disassembler to extract kernel text (code) and the kernel symbol table\textsuperscript{8}. Then, the target symbols (i.e., addresses of kernel functions) are filtered according to the patterns observed. Next, related instructions are located in the kernel text in order to build basic blocks and control flow graphs. Finally, a reachability analysis is performed to discover vulnerabilities. In addition, to improve the accuracy of our analysis, we also introduce a constraint solver into the proposed system, which can be used to find exceptional vulnerabilities.\footnote{Performing privilege escalation is an ideal case, however, in reality, constraints always exist that need to be bypassed.} \footnote{The kernel symbol table can also be fetched by accessing \texttt{/proc/kallsyms}.}

\textsuperscript{7}Performing privilege escalation is an ideal case, however, in reality, constraints always exist that need to be bypassed.

\textsuperscript{8}The kernel symbol table can also be fetched by accessing \texttt{/proc/kallsyms}.
values that might be exploited to bypass some faulty verification.

Figure 4.4 Flow of discovering unknown vulnerabilities

4.3 Implementation and Evaluation

We have implemented a prototype of the proposed KSEFA system. The majority of the system is written in C (i.e., preparation of exploits) and Python (i.e., detection of unknown vulnerabilities). Specifically, we built the latter based on Capstone [13] and Z3 [69]. Capstone is a lightweight disassembly framework that supports multiple platforms and architectures including ARM. We use Capstone as our disassembler. Z3 is a theorem provider that supports many functionalities, including linear real and integer arithmetic. We use Z3 as our constraint solver.

In the following subsections, we first describe the preparation of the evaluation in Section 4.3.1. We then verify the known vulnerabilities and evaluate the effectiveness of the defense mechanisms deployed on the whole in Section 4.3.2. Thereafter, in Section 4.3.3, we introduce two representative unknown vulnerabilities that were discovered by our system. Finally, we measure the response times of different participants by examining when the patches for the firmware images were released.

4.3.1 Evaluation Preparation

Eight known vulnerabilities were collected to perform the evaluation. As mentioned in Section 4.2, they were selected according to the time they appeared in CVE [19], which emerged from 2012 to 2014 (the distributions are 2 in 2012, 4 in 2013, and 2 in 2014, respectively). Table A.1 (see Appendix A) summarizes all of the vulnerabilities collected. The vulnerability source is distinguished, either the Linux kernel or a chipset manufacturer. This reveals the organization in charge, e.g., the Linux kernel community is
responsible for providing official patches for kernel-generic vulnerabilities.

The corresponding date disclosed and community-patch dates are also given. On the one hand, the disclosed date is simply the CVE-ID assigned date, unless there is any other public evidence that can be used to determine the correct date. For example, CVE-2012-6422 was first disclosed on forum xda-developers [91] on 12/15/2012, just 2 days before the corresponding CVE-ID was assigned. As mentioned in the disclaimer [19], the CVE-ID time assigned is just the entry created date, and it does not necessarily indicate the date disclosed. However, it is very difficult (or even impossible) for us to track the exact dates disclosed for all vulnerabilities. Therefore, despite its inaccuracy, we use the CVE-ID assigned date instead, which is not biased sufficiently to skew our observation. On the other hand, we are able to obtain a relatively accurate community-patch date in most cases. For example, the patch date of the Linux kernel development community is always clear. However, the situation becomes a little bit complicated if the chipset manufacturer neither announces any patch publicly nor releases source code. In that case, we have to try to find some related information from third-party sources. Accordingly, the community-patch time calculated here is just a rough estimation rather than a precise measurement.

In our evaluation, we examine nine representative phones (Table 4.3) released between 2010 and 2014 by three popular vendors: Google, Samsung and HTC. The phone model selected either has a tremendous influence and is representative, or has a huge market share. Google’s phones are designed to be reference models for their whole generation, while Samsung has been the market leader for years. Further, if possible, we prefer to use phones that share the same chipset manufacturers with manufacturer-specific vulnerabilities. The release date of a specific device may vary in different regions (with different specifications). For example, the release date of Nexus S [86] in the UK was 12/22/2010, and in Canada, it was 04/01/2011. However, that does not affect our evaluation results, because we focus on the release date of (updated) firmware images, rather than that of the device.

As mentioned in Section 4.2, we need to set up two observation points for each vulnerability, and the second observation point is simply a date some reasonable time after the first one. In our study, we set the community-patch date as the first observation point. The corresponding vulnerability must be known publicly at that moment, and, according to our observations, the patched firmware images have not yet been released. For the second observation point, we believe that three months is a reasonable time for device vendors to release their updates. Thus, in the following, the two observation points are the community-patch date and the date three months after the community-patch date.

For each device, we collected the latest official firmware images released at (or before) the observation points. The corresponding official images were collected manually from the Internet, and included both official and third-party sources [70, 77, 84]. Normally, device vendors provide the latest or stable versions rather than the complete list of all the firmware they have released. Take Google as an example. It provides an official website [36] to download the latest factory firmware images of different AOSP
versions. However, it appears that the unstable versions have been removed. To achieve a fine-grained measurement, we had to collect those removed from the third-party sources. Although we could not guarantee the absolute accuracy of all images collected, most of them are likely to be reliable. In particular, Samsung provides a large number of firmware images by region, and the update time varies across regions. For example, customers in Europe often receive updates faster than do others. Conversely, there is usually a time lag for customers in the US (things become more complicated considering the participation of carriers). To simplify our analysis, we chose Hong Kong as the target region (its Samsung official letter code is TGY) from which to collect all necessary firmware images, because they are provided in a relatively timely manner and are easy to track.

### 4.3.2 Verifying Known Vulnerabilities

The overall results are listed in Table 4.4. Note that the inapplicable cases (labelled as -, either the device is not available at the time point, or the vulnerability is manufacturer-specific and not compatible with the device) are ignored. For a given vulnerability and a device model, the latest firmware images at the two observation points were collected, respectively. Take CVE-2012-4421 and Google’s Nexus 4 as an example. The community-patch date of this vulnerability is 11/15/2012. The device’s latest firmware image at that time was released on 11/13/2012 with build number JOP40C, and this image is vulnerable to that vulnerability. After three months, the latest firmware image at this moment was the one released at 02/15/2013 with build number JDQ39, which is no longer vulnerable.

The percentage of firmware images vulnerable to each vulnerability is shown in Figure 4.5. For a particular vulnerability, the left bar refers to the percentage at the first observation point (i.e., the corresponding community-patch date), and the right bar shows the percentage at the second observation point.
Table 4.4 Overall results with two observation points

<table>
<thead>
<tr>
<th>CVE #</th>
<th>Google</th>
<th>Samsung</th>
<th>HTC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Nexus S</td>
<td>Nexus 4</td>
<td>Galaxy S2</td>
</tr>
<tr>
<td>CVE-2012-4221</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>CVE-2012-6422</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CVE-2013-2094</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>CVE-2013-4738</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CVE-2013-6123</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CVE-2013-6282</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>CVE-2014-0196</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>CVE-2014-3153</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

1: not applicable, either the device is not available at the time point, or the vulnerability is manufacturer-specific and not compatible with the device
2: ×: the firmware image is vulnerable to the vulnerability
3: √: the firmware image is not vulnerable to the vulnerability

point. The proportions vary from 0 (i.e., at the second observation point of CVE-2012-6422) to 100%. On average, the percentage of vulnerable images at the two observation points are 84.78% and 47.83%, respectively. Thus, more than 84% of the latest firmware images are vulnerable to a newly-released, publicly-known vulnerability; even after three months, over 47% of the firmware images are still vulnerable to that vulnerability. Our results also show that kernel-generic vulnerabilities have more impact than do manufacturer-specific ones. Specifically, 85.71% and 54.29% of firmware images, respectively, are vulnerable to kernel-generic and manufacturer-specific vulnerabilities at the first observation point, while 81.82% and 29.27% images, respectively, are vulnerable to kernel-generic and manufacturer-specific vulnerabilities at the second observation point.

Note that there are two cases at the second observation point: 1) the device vendor provides updates but does not patch the vulnerability; 2) the device vendor does not release any updates after three months. Both of these cases provide a relatively large window of times for attackers. Let us look at the most recent vulnerability, i.e., CVE-2014-3153, as an example. The community-patch date is 06/05/2014, and 100% of the latest firmware images are vulnerable. After three months, 88.89% of the firmware images are still vulnerable. In fact, only one device (i.e., HTC One) received the patched updates during the time period.

Figure 4.6a shows the statistical results observed from another dimension, i.e., device vendors. Note that, when measuring different vulnerabilities, if any duplicate firmware images existed, we treat them as different. The performance of Google’s devices (as a whole) is better than that of others, and it is similar to the results obtained from Chapter 3 for app-level vulnerabilities. Samsung’s devices are vulnerable to all compatible vulnerabilities. One may argue that it is probably more reasonable to measure them with kernel-generic vulnerabilities, which are compatible with all devices. However, the results are similar to
In the following, we analyze the results based on the two observation points. Let us focus first on firmware images collected at the first observation points. We know that less than 16% of firmware images are not vulnerable to three related vulnerabilities, including two kernel-generic vulnerabilities (i.e., CVE-2013-6282 and CVE-2014-0196) and one manufacturer-specific vulnerability (i.e., CVE-2013-6123), as follows:

- **CVE-2013-6123**: Nexus 4 and Nexus 5
- **CVE-2013-6282**: Nexus 5
- **CVE-2014-0196**: Nexus S, Nexus 4, Nexus 5 and HTC One

After investigation, we find that both of the two kernel-generic vulnerabilities have been patched in the corresponding devices, whose immunity has nothing to do with the defense mechanisms. The manufacturer-specific vulnerability, CVE-2013-6123 [21], is an improper input validation problem (i.e., of the array index) of Qualcomm’s camera driver. Looking at Nexus 4 (Android 4.4.2; KOT49H) as an example, the key device file (/dev/video100) used to exploit the vulnerability is inaccessible to a normal user, as shown in Figure 4.7a. However, the file permissions remain the same as those for the device with Android 4.3 (JWR66Y), which can be exploited successfully, because at that time it can be
Figure 4.6 Vulnerable firmware images by device vendors

(a) With all compatible vulnerabilities

(b) With kernel-generic vulnerabilities
bypassed easily by acquiring an Android permission associated with the corresponding gid. Figure 4.7b shows the corresponding snippet of /system/etc/permissions/platform.xml, which illustrates the mapping between Android permission and the gid camera.

It is possible for an unprivileged user to be assigned the gid camera by acquiring this permission. Thereafter, the corresponding device file is accessible. Unfortunately, the mapping has been removed since Android 4.4 (KRT16S), which is why this device file can no longer be accessed by unprivileged users. In short, this vulnerability is blocked at the triggering stage of the exploitation.

Next, we focus on the firmware images at the second observation points. Similarly, we examine those that are updated without patching, because they can be used to demonstrate the success of defense mechanisms. Unfortunately, we do not find any such images.

Overall, most of the successful defenses derive from patched firmware images, and thus, they cannot be regarded as an effect of the defense mechanisms.

As discussed in Section 4.2.2, we are always able to measure the effectiveness of defense mechanisms deployed (as a whole) quantitatively by using a mathematical model, which is limited and cannot be used to evaluate defense mechanisms that are used to mitigate kernel information leakage. Apparently, firmware images collected at the first observation points are more meaningful in measuring the effectiveness of defense mechanisms. We also ignore the patched images, which do not help with the measurement. The measurement of effectiveness is shown in Table 4.5. Only two have non-zero val-
ues (i.e., they are effective); however, they are simply the two successful defenses we discussed earlier, which block the vulnerability (i.e., CVE-2013-6123) at the triggering stage. In other words, most of the defense mechanisms deployed (if any) are ineffective. In fact, we also verify the existence of some well-known defense mechanisms (e.g., SEAndroid [80]). Unfortunately, they seem to be useless in preventing these vulnerabilities.

According to our observations, we believe that there are two major reasons for this. Firstly, the mechanisms deployed, which rely on their configuration to be effective, are not configured properly. For example, SEAndroid [80] is, by design, capable of blocking access to the key device file required by a manufacturer-specific vulnerability. However, the only requirement is that it must be configured to do so. Unfortunately, this is not the case in our target firmware images. Secondly, some useful mechanisms that have demonstrated their effectiveness on other platforms (e.g., desktop) have not yet been adopted widely. Of course, due to the constraints of the smartphone environment, some of them may have to be adjusted and others may not be suitable for adoption. In any case, defense mechanisms that are more effective must be configured and deployed properly to mitigate the threats of kernel-level vulnerabilities in commodity Android devices.

### Table 4.5 Measurement of effectiveness

| CVE #       | \(|T|, |E|, |L|\) | Google |                     | HTC          |
|-------------|-----------------|--------|---------------------|--------------|
|             | \(\{\text{Nexus S}, \text{Nexus 4}, \text{Nexus 5}, \text{Galaxy S2}, \text{Galaxy S3}, \text{Galaxy S4}, \text{Sensation}, \text{One X}, \text{One}\}\) |        |                     |              |
| CVE-2012-6123 | \(1, 3, 2\) | -1     | 0                   | 0             | 0            | 0             | -             | -             | -             |
| CVE-2012-6122 | \(1, 5, 2\) | -1     | 0                   | 0             | 0             | 0             | 0             | 0             | 0             |
| CVE-2013-2094 | \(1, 3, 2\) | 0       | 0                   | 0             | 0             | 0             | 0             | 0             | 0             |
| CVE-2013-4738 | \(1, 1, 2\) | -1     | 0                   | -             | -             | -             | 0             | -             | 0             |
| CVE-2013-6123 | \(1, 3, 2\) | -1     | 1                   | 1             | -             | -             | 0             | -             | 0             |
| CVE-2013-6282 | \(2, 5, 2\) | 0       | 0                   | 0             | 0             | 0             | 0             | 0             | 0             |
| CVE-2014-0196 | \(1, 3, 2\) | -1     | 0                   | 0             | 0             | 0             | 0             | 0             | -             |
| CVE-2014-3153 | \(1, 5, 2\) | 0       | 0                   | 0             | 0             | 0             | 0             | 0             | 0             |

1: Not applicable, either not compatible, or the vulnerability has been patched.

### 4.3.3 Discovering Unknown Vulnerabilities

Our system has already discovered several unknown vulnerabilities in devices that use chipsets of MediaTek (MTK) [58]. MTK has grown rapidly in recent years, and since 2013, it has become the second largest chipset vendor [57] in the world. MTK has been the dominant in China [17] for years, and many manufacturers in China adopt MTK’s solution to build their devices. Considering the large numbers of smartphones that are sold in China (e.g., 107.5 million in the 4th quarter of 2014 [51]), the influence
of MTK’s vulnerabilities is undoubtedly great. In the following, two representative vulnerabilities are introduced to demonstrate the effectiveness of our proposed system.

FDVT_ioctl has been built!
FDVT_ioctl
- > [167]0xc0040544: eb
- > [73]0xc003f894: e5830000 str r0, [r3] <<<< 

Figure 4.8 Snippet of execution result of /dev/camera-fdvt

The first vulnerability is quite straightforward. It is related to a device file called /dev/camera-fdvt. Through the corresponding ioctl interface, it is able to write arbitrary kernel memory addresses with arbitrary data. Figure 4.8 shows the vulnerability path found by our system, which indicates that there is a vulnerable path from function FDVT_ioctl to function MT6573FDVT_SetRegHW. The memory write takes place at the 73rd instruction of the function MT6573FDVT_SetRegHW, where <<<<< labels the associated registers. In particular, when labeling the branch instruction, it describes the registers that are associated with the parameters.

The second vulnerability can be exploited through the corresponding mmap interface. It is a bit complicated because of the presence of input verification. Specifically, it checks two input parameters (i.e., the length and the offset) when invoking mmap system call. In this case, we have to rely on the constraint solver to provide the details. Figure 4.9 shows a snippet of the results. We can see that in order to reach the invocation of remap_pfn_range, registers r2, r3 and r7 must satisfy the corresponding requirements. In particular, r7 ≥ 0x80000000 shows the evidence of an integer conversion overflow.

4.3.4 Response Time

The community-patch time of each vulnerability is shown in Figure 4.10a; the average is 74 days. The average community-patch time of kernel-generic vulnerabilities is less than that of manufacturer-specific ones, i.e., 65 days vs. 83 days. Note that the community-patch time of CVE-2014-0196 is 151 days, which seems to be an exception. Thus the average community-patch time of kernel-generic vulnerabilities is only 36 days, when CVE-2014-0196 is excluded, at which point the corresponding overall average community-patch time is 63 days. In any cases, the community of the Linux kernel reacts faster than do the chipset manufacturers.

Table 4.6 lists the vendor-patch time for each device, while Figure 4.10b shows the average vendor-
ISP_mmap has been built!
ISP_mmap
-> [134]0xc003ae0c: eb03eb01 bl #0xc0135a18 ;remap_pfn_range <<<<<< {
[2]: [1]} {
[3]: [1]}
-> [39]0xc003ac90: eb03eb60 bl #0xc0135a18 ;remap_pfn_range <<<<<< {
[2]: [1]} {
[3]: [1]}

[Or(r2 != 16),
Or(r3 == 352354304, r3 == 352321536, r3 == 268435456, r3 == 268455936),
Or(r7 <= 4096, r7 <= 65536, r7 <= 12288, r7 <= 16384)] sat
r7
-> 0x0 sat
-> 0x10000000 unsat
-> 0x20000000 unsat
-> 0x30000000 unsat
-> 0x40000000 unsat
-> 0x50000000 unsat
-> 0x60000000 unsat
-> 0x70000000 unsat
-> 0x80000000 sat
-> 0x90000000 sat
-> 0xa0000000 sat
-> 0xb0000000 sat
-> 0xc0000000 sat
-> 0xd0000000 sat
-> 0xe0000000 sat
-> 0xf0000000 sat

Figure 4.9 Snippet of execution result of /dev/camera-isp

patch time for each device to fix at least one of the vulnerabilities collected. Note that it is possible that some devices do not have the corresponding updated firmware image for a given vulnerability: either the device is no longer supported, or the vulnerability is inapplicable to the device. The extreme case is Google’s Nexus S, which has no updated firmware images for any of the vulnerabilities collected. Thus its average vendor-patch time is inapplicable. For those with at least one applicable updated image, we can calculate the overall average vendor-patch time for different vendors. The average vendor-patch time for Google’s devices is 189 days, which is more than Samsung’s 144 days and HTC’s 145 days. The overall average vendor-patch time is 156 days, which is more than twice the average community-patch time (i.e., 74 days or 63 days excluding the exceptional case).

It is far from timely for ordinary customers to receive corresponding security updates due to the slow update cycle (the entire patch time is 230 days). To make matters worse, the latency may be prolonged by the participation of carriers. In addition, we might be pessimistic about the overall circumstances, because the devices we collected are quite popular, so the treatment is usually better than other models.
Figure 4.10 Community-patch time and vendor-patch time
Table 4.6 Vendor-patch time

<table>
<thead>
<tr>
<th>CVE #</th>
<th>Google</th>
<th>Samsung</th>
<th>HTC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Nexus S</td>
<td>Nexus 4</td>
<td>Nexus 5</td>
</tr>
<tr>
<td>CVE-2012-4221</td>
<td>-</td>
<td>113</td>
<td>-</td>
</tr>
<tr>
<td>CVE-2012-6422</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CVE-2013-2094</td>
<td>-</td>
<td>184</td>
<td>-</td>
</tr>
<tr>
<td>CVE-2013-4738</td>
<td>-</td>
<td>142</td>
<td>-</td>
</tr>
<tr>
<td>CVE-2013-6123</td>
<td>-</td>
<td>231</td>
<td>231</td>
</tr>
<tr>
<td>CVE-2013-6282</td>
<td>-</td>
<td>26</td>
<td>-</td>
</tr>
<tr>
<td>CVE-2014-0196</td>
<td>-</td>
<td>182</td>
<td>-</td>
</tr>
<tr>
<td>CVE-2014-3153</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

1: Not applicable, either the device does not use the vulnerable chipset, or the vulnerability has been fixed before releasing the device.

that are not as popular. In short, even known vulnerabilities are still dangerous. It is fair for customers to urge each participant in the ecosystem, device vendors in particular, to shorten the corresponding patch time.

4.4 Discussion

In considering the overall results of vulnerable firmware images, we believe that our results are conservative for two reasons. Firstly, there are a number of old devices (even flagship ones) whose updates are no longer supported. For example, HTC One X, although the flagship device of HTC, will not receive any further updates after those in 4.2.2 [42]. Further, although some popular devices are supported continuously, e.g., Samsung’s Galaxy S3 GT-I9300 (International version), which is older than HTC One X and has already received 4.3 updates ⁹, the situation overall cannot be regarded as optimistic. Secondly, there are also a large number of devices that cannot receive the updates provided timely due to certain limitations (e.g., network issues). Therefore, the real percentage of vulnerable firmware images is definitely higher than our results suggest.

In the following, we would like to acknowledge some of the limitations of our work. Firstly, our collection of vulnerabilities is hardly exhaustive. The ideal collection of vulnerabilities is able to meet all aspects of the analysis, but in practice, this is not the case. For example, if enough chipset related vulnerabilities could be collected, then we would be able to perform a comparison study to evaluate the security of different chipset vendors. However, although some vendors do make an effort (e.g., Qualcomm provides security advisories on its CodeAurora forum [68], which gives us relatively precise and timely information), not all of them are eager to share such related information with the public.

⁹Actually, the variants can be updated to 4.4
Secondly, the collection of firmware images is unlikely to be very accurate. Due to the sales region the device vendors made (e.g., Samsung), it is very difficult to determine actual representative update images for a device model. In particular, for a given region, it is possible that some updates are timely while others lag. Therefore, from our experience, we believe that using Hong Kong as a base is a relatively good choice. Lastly, there are two primary concerns in the discovery of unknown vulnerabilities. On the one hand, the false positive rate is high in the following two cases: 1) too many memory write instructions, and 2) too many branches. Thus, to improve accuracy, we need to find ways to prune the infeasible paths. On the other hand, the detection only covers manufacturer-specific vulnerabilities. In our experience, it is difficult to infer patterns in kernel-generic ones that can be used to direct detection. Traditional kernel-generic vulnerabilities (e.g., stack overflow) are unlikely to remain after decades of community audits. Others, like CVE-2014-0196, are often associated with certain race conditions, which makes the matters even worse. Thus, it is likely that a dynamic analysis is more suitable in identifying such vulnerabilities.

4.5 Summary

In this chapter, we evaluated the impact of known kernel-level vulnerabilities in commodity Android devices and studied the effectiveness of defense mechanism deployed to mitigate threats. We conducted an evaluation of nine devices from three different vendors: three models (three models each; actually, three different generations). We first evaluated the overall status of the vulnerable firmware images. On average, 84.78% of the latest firmware images were vulnerable to a newly publicly disclosed vulnerability, and 47.83% of the latest (updated) images were still vulnerable after three months. These results can also be used to measure the effectiveness overall of the defense mechanisms deployed, i.e., most are not as effective as expected. Furthermore, based on the insights obtained from the study of known vulnerabilities, our proposed system was also able to discover zero-day vulnerabilities by performing a static disassembly-level data-flow analysis. We discovered several such vulnerabilities, and the findings demonstrated the effectiveness and promise of the proposed system. Lastly, we also measured the response time of each participant in providing patches for the corresponding vulnerabilities. Our results showed that the average time taken to release the official patched images (starting from the disclosed date of the corresponding vulnerability) was 230 days, a large time window for attackers.
Chapter 5

Mitigation of App- and Kernel-Level Vulnerabilities

In Chapters 3 and 4 we discussed the detection of app- and kernel-level vulnerabilities, respectively. In the following, we propose a defense mechanism that can be used to mitigate the threats introduced by the aforementioned vulnerabilities.

5.1 Introduction

As we have demonstrated in the previous chapters, current defense mechanisms are not as effective as expected. We have also shown that there is usually a significant time window (i.e., more than 6 months) available to attackers before end users receive official updates. In reality, the security updates provided by device vendors are usually accompanied with other kinds of updates (e.g., UI), which not only enlarges the package size, but also slows down the update cycle. In short, better solutions are needed to address these problems.

However, it is not easy to design an effective defense system to mitigate the threats posed by both the app- and kernel-level vulnerabilities. Firstly, it is quite difficult to find common patterns to cover all of these diverse vulnerabilities. Secondly, these vulnerabilities can be exploited by any unprivileged app, the behavior of which cannot be confined easily. Furthermore, it is necessary to design a solution that can provide users with a quick response.

In this chapter, we are interested in exploring approaches to allow for runtime system patching. Specifically, it allows us to focus on fixing the vulnerabilities rather than designing and developing generic but complicated defense mechanisms to cover various sorts of vulnerabilities. Researchers have published several works on patching systems. For example, KSplice [5] is capable of applying patches...
(no significant semantic changes to Linux kernel data structures) for desktop PCs at runtime without rebooting the OS. PatchDroid [60] is a system that can be used to patch framework-level vulnerabilities in Android devices. AppSealer [96] is designed to generate security patches for component hijacking vulnerabilities in Android apps. However, these target neither app- nor kernel-level vulnerabilities. FireDroid [74] is a policy-based security framework that, is capable of controlling the behavior of malicious apps by intercepting system calls using the ptrace mechanism. Although it is able to patch vulnerabilities dynamically, it has the following limitations. Firstly, the requirement to modify the firmware image seriously affect the feasibility of deployment. Secondly, FireDroid is unsuitable for monitoring system calls with heavy workloads (e.g., mmap), as it is based on ptrace, the performance overhead of which has been demonstrated to be a great concern [45]. Lastly, FireDroid is not easy to use to address app-level vulnerabilities. Specifically, it is necessary to transfer the understanding of vulnerabilities into the knowledge of corresponding system calls. Despite these limitations, the idea of preventing vulnerabilities with a policy-based security system is still valuable.

In this chapter, we propose a novel, transparent, policy-based Dynamic Vulnerability Mitigation Framework for Android (DVMFA) to protect vulnerable devices. DVMFA is a context-aware system that focuses on both app- and kernel-level vulnerabilities, and as the name suggests, it require no modification or recompilation of pre-loaded apps or device firmware images. In contrast to FireDroid, DVMFA addresses app- and kernel-level vulnerabilities simultaneously. Specifically, we first design a policy language for kernel-level vulnerabilities as the base set, and then extend it to support app-level vulnerabilities with richer semantic specification and bindings. However, the method used to perform the patching is different, i.e., app-level vulnerabilities are patched by intercepting corresponding Android APIs with an app-level monitoring module, while kernel-level vulnerabilities are patched by intercepting related system calls with a LKM (i.e., loadable kernel module).

We have implemented a DVMFA prototype, and conducted an evaluation to measure its effectiveness, portability, and performance. The evaluation results demonstrate that DVMFA is able to prevent all vulnerabilities we studied in Chapters 3 and 4, and it can be deployed in most of the devices we evaluated. The results also show that the performance overhead is negligible (with less than 3% slowdown). Thus, DVMFA is quite suitable as a temporary solution to provide users with a quick response, and shrink the time window open for exploitations. To protect their products, the device vendors could also adopt and deploy DVMFA as a built-in module, which would make it more feasible and effective.

The rest of this chapter is organized as follows. We present our methodology and system framework in Section 5.2, and describe the results of the implementation and evaluation with case studies in Section 5.3. We then discuss potential improvements in Section 5.4. Finally, we provide a chapter summary
in Section 5.5.

## 5.2 Design

Figure 5.1 summarizes the overall architecture of the proposed DVMFA system, which has three major components: policy manager, API monitor and syscall monitor. If we have an in-depth knowledge about the vulnerabilities of interest, then we are able to generate corresponding policies for them by using the policy language we defined. The policies generated, including those at the app- and kernel-levels, are fed into the policy manager. Thereafter, the API and syscall monitors first hook the target APIs and syscalls designated by those policies, and then verify the input from the suspicious sources (apps and executable binaries) to the vulnerable Android components and system calls. Finally, the policy actions are enforced as necessary. In the following, we will first introduce the definition of our policy language in Section 5.2.1, and then describe the API and syscall monitors in Sections 5.2.2 and 5.2.3, respectively.

![Figure 5.1 The overall architecture of DVMFA](image)

### 5.2.1 Policy Language

Listing 5.1 shows the meta-definition of the policy language. We try to make the syntax of the language consistent with widely accepted pseudo programming languages that are used to describe algorithms. Basically, “#” is the beginning of the comment line; “:=” is the meta definition. The meanings of some
operators and signs are defined as follows:

- **Brackets and Braces** the angular brackets "<>" designate a must-have item, and curly braces "{}" indicate an optional item;
- **Assignment operator** sign "=" is used to assign the value of the right side to the left side;
- **Logical operator** sign "|", "&" and "∼" represent the logic OR, AND and NOT;
- **Comparison operator** "==", "!=", "<", "≤", ">" and "≥";
- **Misc** ":" refers to a followed-by relationship, i.e., A : B means A must be followed by B; "@" and "@" are the delimiters of description of parameters. Note that "@" can also be used as the indicator of constraint in app-level policies or command in kernel-level policies.

For both app- and kernel-level policies, the basic structure is the same, i.e., policy name, followed by the vulnerability description of the vulnerable components and parameters, and finally an action. A valid policy description starts with a policy name enclosed in square brackets. The formats of the following two parts rely on the type of vulnerability (i.e., app- or kernel-level).

### Listing 5.1 Meta definition of policy language

```plaintext
# ' ' # := beginning of the comment; policy_type := <app|kernel>
[<policy_name>]
 if policy_type is app
 # type := <java.lang.String | android.content.Intent|...> 
 # constraint := <internal fields: names and types>
 HOOK_PKGNAME=<pkg−name>
 HOOK_CLASSNAME=<class−name>
 HOOK_CLASSTYPE=<ACTIVITY | RECEIVER | SERVICE | PROVIDER | ... >
 HOOK_METHODNAME=<method−name>
 HOOK_METHODPARAMS=<param1>,<param2> ... .
 METHODHOOKPARAM=<index>:<type>:<name>{@constraint}
 ACTION=<action−type>:<original−value>:<new−value>
 else if policy_type is kernel
 # type := <uint32 | string |...> 
 # constraint := <bound | pattern |{:compare with other parameters}>
 DEVNAME=<dev−name>
 HOOKTYPE=<syscall | security >
 HOOKNAME=<syscall−name | security −name>
 HOOKPARAMS={@command}<param1: type: constraint>,<param2: type: constraint> ....
 ACTION=<deny | warn | allow>
```

62
For app-level vulnerabilities with rich and high level semantic information, the corresponding description first indicates the signature of the vulnerable method, including the package, class, and method names, and the method parameters. After that, a precise description of the vulnerable parameter is given as the input. For kernel-level vulnerabilities with low-level semantic information, the description is relatively simple. Only the names of the target system call and corresponding vulnerable parameters are required, as it is simple to hook a system call uniquely by searching the system call table. Sometimes, vulnerable system calls (especially manufacturer-specific ones) are tied to specific devices, which can be designated with `DEVNAME`. In addition, DVMFA also supports the hooking of existing security modules, such as SELinux, which can be specified with `HOOKTYPE`. We know that SELinux provides a good MAC control for many system calls, and SEAndroid [80] (SELinux for Android) has become the built-in security module in recent Android versions. Thus, in order to mitigate kernel-level exploits, it seems appropriate to build our system based upon SELinux; however, the following limitations restrict its feasibility. Firstly, the semantics of the policy is too limited to describe the vulnerabilities (e.g., it is impossible to describe the valid input range of the parameters); Secondly, it has incomplete coverage of system calls, and some of the vulnerabilities we studied cannot be handled by extending SELinux alone (e.g., CVE-2014-3153). Therefore, we have to hook system calls directly, but we still are able to provide the support to hook security modules, which may be helpful when porting our solution as an extension with the further development of SELinux.

Note that parameters for app- and kernel-level vulnerabilities are described differently, which leads naturally to different formats of `ACTION`. For app-level vulnerabilities, the critical values for exploiting vulnerabilities are usually unique, such as a specified extra value stored in an intent, while the feasible values for exploiting kernel-level vulnerabilities are often within a range, such as the size of the input buffer. In addition, all of the in-depth details can be specified with the corresponding `constraint`. With respect to `ACTION`, for app-level vulnerabilities, we can modify the specific values with predefined invalid values that can be captured by the vulnerable components; at the kernel-level, an `EINVVal` value is always returned. In any case, we can specify any other necessary behaviors to meet our requirements. Finally, although the formats of `ACTION` for the two types of vulnerabilities differ, the final results are either `deny` (by default), `warn`, or `allow`.

5.2.2 API Monitor

The API monitor module is used to intercept behaviors that may trigger app-level vulnerabilities. On startup, it first loads app-level policies provided by the policy manager. After parsing out all policies, it hooks corresponding target APIs accordingly, so that the new input flowing into the vulnerable components can be verified and blocked (or modified), if necessary. The majority of the types of triggering
procedures for app-level vulnerabilities are related to either intent sending (activity, receiver, and service) or uri accessing (content provider). Thus, we have to hook the places where suspicious intents and uris can be located and verified. Obviously, the entrypoint of one vulnerable component is the hooking point we need. For example, if the vulnerable component is an activity, then the hooking point will be the onCreate method of that activity. We are able to verify uri by examining operations in ContentResolver (e.g., query).

However, we also must distinguish the suspicious requests from those that are legitimate and valid. For example, we cannot block legitimate requests that may be initiated by other pre-loaded apps to those vulnerable components. It is necessary to find a way to divide requests based on their provenance. Of course, one app's uid can be used as a heuristic filter, i.e., apps with uid greater than 10000 can be regarded as suspicious. However, sometimes a pre-loaded app may also be assigned a uid greater than 10000. Recall our assumption that pre-loaded apps may be vulnerable but never malicious. Thus, a practical solution is to divide apps into different groups at the start, and label the intents originating from the suspicious apps with extra information. Specifically, the trustworthy (pre-loaded) apps are put on a white list, and only suspicious (third-party) apps are labelled.

The challenge is to find the appropriate place to perform the labeling. We can define the component that sends the intent as the source component, and the destination component as the target component. As shown in Figure 5.2, internally, an intent is sent first from the source component to the Android framework. After parsing out that intent, the framework assembles a new intent and forwards it to the target component. However, it is impossible to retrieve information about the source component from the new intent generated. Apparently, the labeling has to be accomplished before the new intent is assembled. We have observed that the method writeToParcel in android.content.Intent is the critical point at which to parse out intents for all components. Thus it is also the ideal hooking place for labeling. Note that uri does not need such labeling, because we can block it at the beginning, i.e., from the source component to the Android framework, which provides us with sufficient information to perform the verification.

Apart from intent and uri, there are other types of input that needed to be verified. For example, one vulnerability is triggered by communicating vulnerable component with a socket (i.e., android.net.LocalServerSocket). The exploitation cannot be intercepted with the methods mentioned earlier because it has nothing to do with intent or uri. Thus, these types of vulnerabilities are handled as special cases, as are the descriptions of the corresponding policies. In the previous example, we define the policy by the type of socket, and hook the constructor of android.net.LocalServerSocket to examine the name of the socket. One may argue that hooking the socket can be accomplished by hooking system call open. This is true; however, it does not necessarily mean that it is a good choice. Firstly, as mentioned earlier, in order to maintain the distinction between app-level and kernel-level procedures,
we try to avoid handling one type of vulnerability with semantics from another type. Further, as we show in 5.2.3, we try to avoid (or reduce if not possible) monitoring system calls such as `open`, which may be invoked quite frequently.

### 5.2.3 Syscall Monitor

The syscall monitor module is designed to intercept system calls, which are the entrypoints for kernel-level vulnerabilities. The basic idea is to hook the system call table by using the LKM. Although not all devices support the LKM (e.g., Google’s Nexus series) \(^2\), most popular devices from various device vendors (e.g., Samsung) do provide such support, which should not affect the deployment greatly.

By comparison with system call hooking in traditional desktop PCs, there are several problems that must be solved to make the LKM work properly on smartphone platforms. One major challenge is the fact that in many, if not most, cases, it is impossible to obtain the same environment (e.g., kernel source) to build a LKM with the appropriate format, i.e., one that is able to bypass the format restriction mechanism adopted by the kernel. Specifically, two restrictions have to be bypassed in most cases [15]: 1) vermagic verification, and 2) `module_layout` (or `struct_module` in kernel version 2.6.x) CRC checksum verification if `CONFIG_MODVERSIONS` is enabled. Both vermagic \(^3\) and the CRC checksum of `module_layout` can be found [15] in existing kernel modules \(^4\). We are able to replace the corresponding values with those retrieved to bypass all of the verification. In addition, the CRC checksum of all referenced functions will be examined by the kernel; this can be avoided by 1) searching kernel symbols to locate

---

\(^2\) In this case we have to modify the kernel configuration to build a new kernel image and generate the new firmware, which is complicated than the similar processing mentioned in [74].

\(^3\) vermagic can also be fetched from the kernel image.

\(^4\) i.e., `.ko` files, normally can be found in folders such as `/system/lib/modules` or `/lib/modules`
addresses of kernel functions (e.g., printk), and 2) implementing functions (e.g., memset) manually. Sometimes, device vendors may introduce their own restrictions. For example, Samsung’s devices may adopt the lkmauth mechanism if CONFIG_TIMA_LKMAUTH is enabled. Although inelegant, such mechanisms can be bypassed by modifying the corresponding instructions in kernel text through /dev/kmem or /dev/mem.

Another challenge arises from the fact that some important kernel data structures (e.g., task_struct) may vary in different kernel versions, or even within the same version but with different kernel configurations. Worse still, we know that device vendors’ customization may introduce modifications of some kernel data structures. Under the circumstances, we cannot use the pre-compiled structures of the LKM to refer their members directly. Instead, we have to calculate the corresponding offsets dynamically by probing the runtime information according to heuristic patterns.

Lastly, we have to find a way to locate the system call table and security modules, such as SELinux. The former can be retrieved by examining the instructions of vector_sw [15], while the latter can be determined by locating the addresses of symbols (i.e., security_ops and default_security_ops) from a kernel function called reset_security_ops.

5.3 Implementation and Evaluation

We have implemented a DVMFA prototype as a mix of Java code and C code with 1920 and 3374 lines of code (LOC), respectively. We built the API monitor module of app-level based on Xposed [72], a framework that helps build Android API monitor modules by providing uniform interfaces. Xposed can hook all apps by modifying app.process, which facilitated our work greatly and allowed us to focus on methods to patch vulnerabilities and their corresponding policy descriptions. We had to build the syscall monitor module for kernel-level vulnerabilities by hand, although we did find suggestions on how to do so in some references (e.g., [15]). Note that in our current prototype, the ACTION is always deny, because we have already verified the vulnerabilities in the previous chapters. Thus, all malicious attempts are prevented explicitly by our system.

Listings 5.2 and 5.3 give concrete examples of the policy description of an app- and a kernel-level vulnerability, respectively. Specifically, Listing 5.2 describes an app-level vulnerability that may lead to system rebooting without any confirmation from the user. It can be triggered by sending a specified intent to start an activity called com.sec.app.RilErrorNotifier.PhoneCrashNotifier. As shown in Listing 5.2, the index of the METHODHOOKPARAM is -1, which is a special case, because the valid index begins from 0 (i.e., it follows the convention of the C and Java programming languages). By definition, this indicates that we should pay attention to the internal data of the variable (with type android.app.Activity), rather than treating it as the parameter list. In particular, if the intent obtained
from the variable has a string-type extra named `message` with the paired value “`DeviceReset`”, it has to be replaced with another value “`Unknown`”. Listing 5.3 shows the policy description of the zero-day vulnerability we discovered in Section 4.3.3. Specifically, this vulnerability is related to the device file `/dev/camera-isp`, as indicated by the assignment of `DEVNAME`. The vulnerable system call is `sys_mmap`. `HOOKPARAMS=1:uint32:0:0x7fffffff` indicates that the valid range of the second parameter with type `uint32` is 0 to 0x7fffffff, inclusively.

**Listing 5.2** An example of policy description of app-level vulnerability

```plaintext
[PhoneCrashNotifier_REBOOT]
HOOK_PKGNAME=com.sec.app.RilErrorNotifier
HOOK_CLASSNAME=com.sec.app.RilErrorNotifier.PhoneCrashNotifier
HOOK_CLASSTYPE=ACTIVITY
HOOK_METHODNAME=onCreate
HOOK_METHODPARAMS=android.os.Bundle
METHODHOOKPARAM=-1:android.app.Activity:android.content.Intent@extra@java.lang.String@message
ACTION=REPLACE:DeviceReset:Unknown
```

**Listing 5.3** An example of policy description of kernel-level vulnerability

```plaintext
[mmap−camera−isp]
DEVNAME=/dev/camera−isp
HOOKTYPE=syscall
HOOKNAME=sys_mmap
HOOKPARAMS=1:uint32:0:0x7fffffff
ACTION=deny
```

In our evaluation of DVMFA, we demonstrate the following three aspects:

- effectiveness: ability to mitigate exploits;
- portability: ability to be deployed on different devices, and

Table 5.1 summarizes the patchable vulnerabilities of our current DVMFA prototype. We are able to patch all vulnerabilities we studied in Chapters 3 and 4. Table 5.2 shows the portability of DVMFA. As the official Xposed works only with root access on Android 4.0.3 or later [72], we focus on devices with stock firmware images of the required versions. Thus the only limitation is the support of the LKM. As discussed in Section 5.2.3, Google’s Nexus series does not support the LKM, and the only remedy is to rebuild the kernel image, i.e., by enabling option `CONFIG_MODULES` of the kernel configuration.
### Table 5.1 Patchable vulnerabilities of current DVMFA

<table>
<thead>
<tr>
<th>Vulnerability Type</th>
<th>Name</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>app-level</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>capability leak</td>
<td>CALL_PRIVILEGED</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>MASTER_CLEAR</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>REBOOT</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>RECORD_AUDIO</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>SEND_SMS</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>SHUTDOWN</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>WRITE_SMS</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>OTHER</td>
<td>✓</td>
</tr>
<tr>
<td><strong>kernel-level</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>manufacturer-specific</td>
<td>CVE-2012-4221</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>CVE-2012-6422</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>CVE-2013-4738</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>CVE-2013-6123</td>
<td>✓</td>
</tr>
<tr>
<td>kernel-generic</td>
<td>CVE-2013-2094</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>CVE-2013-6282</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>CVE-2014-0196</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>CVE-2014-3153</td>
<td>✓</td>
</tr>
</tbody>
</table>

1, ✓: it means this type of vulnerability can be patched by current DVMFA prototype.

Fortunately, all the other devices we examined (including devices that are not listed in Table 5.2) support the LKM; thus, the deployment of DVMFA is feasible in most cases.

In Sections 5.3.1 and 5.3.2, respectively, we first describe detailed case studies of the API and syscall monitor modules, and then evaluate their performance in Section 5.3.3. The case studies are conducted using a Samsung Galaxy S3 device running Android 4.0.4 with Linux kernel 3.0.15. This is one of the top devices and, among all the devices we examined, it has the largest number of the app- and kernel-level vulnerabilities we studied. For the performance evaluation, besides Samsung Galaxy S3, we examine two other devices: Huawei G700 and Samsung Galaxy S4, which run with Android versions 4.2.1 and 4.4.2, respectively. These three make our coverage of the performance evaluation more representative (i.e., from 4.0.4 to 4.4.2).

#### 5.3.1 API Monitor Module

By default, our API monitor targets all pre-loaded apps, but it does not label intents initiated by all pre-loaded apps. As we discussed earlier, we assume that a pre-loaded app may be vulnerable, but not mali-
Table 5.2 Portability of DVMFA

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Model</th>
<th>Android Version</th>
<th>Portability</th>
<th>API Monitor</th>
<th>Syscall Monitor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google</td>
<td>Nexus 4</td>
<td>4.2</td>
<td>√</td>
<td>√</td>
<td>×</td>
</tr>
<tr>
<td>Google</td>
<td>Nexus 5</td>
<td>4.4.4</td>
<td>√</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>HTC</td>
<td>One X</td>
<td>4.0.4</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>HTC</td>
<td>One (2013)</td>
<td>4.1.2</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Samsung</td>
<td>Galaxy S3, GT-I9300</td>
<td>4.0.4</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Samsung</td>
<td>Galaxy S4, GT-I9502</td>
<td>4.4.2</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Huawei</td>
<td>G510</td>
<td>4.0.4</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Huawei</td>
<td>G700</td>
<td>4.2.1</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Xiaomi</td>
<td>MI3</td>
<td>4.2</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>Xiaomi</td>
<td>MI4</td>
<td>4.4</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
</tbody>
</table>

1 √: applicable.
2 ×: not applicable.

We also provide an interface for the user to enable or disable the target pre-loaded apps (as shown in Figure 5.3a), so that users can control the interception; the changes saved become effective after rebooting. Upon detection of a malicious attempt, we block the access directly and warn the user, as shown in Figure 5.3b. The warning message describes the source and the behavior of the malicious attempt. For example, Figure 5.3b states that a component called com.sec.app.RilErrorNotifier.PhoneErrService has received a malicious intent from an app with package name com.example.samsung.demo. The action of the intent is android.intent.action.REFRESH_SIM_RESET.

The malicious attempt must be blocked, but the benign one should be allowed. The red rectangle in Figure 5.4a simply shows the log of the previous malicious attempt. More details (i.e., the calling package name, class name and uid) are provided. The blue rectangle in Figure 5.4a gives an example of the benign attempt, which is actually the normal operation of rebooting the device. The user first presses the power button and then selects the restart option, and finally confirms the action explicitly, as shown in Figure 5.5. Note that because the intent is initiated by a trustworthy component without labeling, the corresponding labeling details are invalid (i.e., either null or -1).

Figures 5.3c and 5.4b show the warning message blocking uri accessing and the corresponding log, respectively. They tell us that a component called com.sec.android.sCloudBackupProvider has received a malicious request to access uri content://com.sec.android.sCloudSMSBackupProvider/sms.
5.3.2 Syscall Monitor Module

We use CVE-2014-3153 [23] to demonstrate the effectiveness of the syscall monitor module. This is a kernel-generic vulnerability derived from the Fast User space muTEX (futex) subsystem, which is the basis of several mutual exclusion mechanisms (e.g., pthread) in Linux. 5

The vulnerable function, `futex_requeue`, (through kernel 3.14.5) does not ensure that calls have two different futex addresses (i.e., addresses of aligned integer variables in user space), that allow local users to gain privilege via some crafted commands [23]. Figure 5.6a shows the execution results of our exploit in the vulnerable device. To make the privilege escalation clearer, we executed the command `id` before and after the execution of the exploit (i.e., the red rectangles in Figure 5.6a). It is easy to see that both the uid and gid were changed from 2000 (i.e., `shell` in Android) to 0 (i.e., `root`), which means the exploit succeeded in gaining the root privilege.

In order to patch it, we have made a policy to verify the two addresses, as shown in Listing 5.4. In particular, this vulnerability has nothing to do with any device file; thus, the assignment of `DEVNAME` is left

---

5 Although Linux kernel provides a `futex` system call interface, users do not invoke it directly in most cases.
The vulnerable system call is `sys_futex`. HOOKPARAMS=`@FUTEX_CMP_REQUEUE_PI:0:uint32:::!=4` indicates that if the command (or operation code) is FUTEX_CMP_REQUEUE_PI (i.e., which specifies the invocation of the vulnerable function `futex_requeue`), the value of the first parameter with type `uint32` shall not be equal to the value of the fifth parameter. Figures 5.6b and 5.7 show the execution results after installing the syscall monitor and the log to block the malicious attempt by enforcing the policy, which shows that the exploit could no longer obtain the root privilege. Note that the exploit is forced to inform us of the presence of the vulnerability, as indicated by the blue rectangles in Figures 5.6a and 5.6b, respectively. The latter demonstrates that this vulnerability has been patched by our system.

Listing 5.4 Policy description of CVE-2014-3153

```
[CVE-2014-3153]
DEVNAME=  
HOOKTYPE=syscall
HOOKNAME=sys_futex
HOOKPARAMS=@FUTEX_CMP_REQUEUE_PI:0:uint32:::!=4
ACTION=deny
```
5.3.3 Performance Evaluation

To evaluate the effect of the prototype on performance, we conduct benchmark-based measurements on three devices supported: Galaxy S3 and Galaxy S4 from Samsung, and Huawei G700. We use several benchmarks to evaluate the performance overhead of our proposed system. For each benchmark, we measure the performance in two different scenarios: 1) baseline refers to the results obtained from the device without DVMFA, and 2) busy refers to results from the device with DVMFA enabled. All of the results are normalized with those from the baseline to reveal the possible performance overhead introduced by DVMFA. The results are the average values of ten different executions.

We first use NenaMark2 [62] and SunSpider [81] (version 1.0.2) to evaluate the overall performance, i.e., NenaMark2 for GPU-intensive workloads, and SunSpider for CPU/IO-intensive workloads, respectively. As shown in Figure 5.8, DVMFA incurs negligible effect on system performance (with less than 1% slowdown). We also use another comprehensive benchmark, Antutu[3], to evaluate performance. As shown in Figure 5.9, the performance overhead is still negligible (less than 3%). In comparison, for Samsung Galaxy S3, the performance overhead is as follows:

- **CPU FP** 2.08%
(a) Attempt to exploit CVE-2014-3153

(b) Attempt to exploit CVE-2014-3153 after installing syscall monitor

Figure 5.6 CVE-2014-3153: exploitation and mitigation

Figure 5.7 Log of blocking CVE-2014-3153
• RAM Speed  2.15%
• Storage I/O  2.96%
• Database I/O  0.87%

Note that the performance overhead of some tests (e.g., 2D) is negative. We believe there are two possible reasons for this: 1) the test has little to do with the proposed system, or 2) the performance overhead is too small to be evaluated accurately by the benchmark. In any case, the results demonstrate that the performance overhead is negligible.

5.4 Discussion

The proposed policy language is able to cover both app- and kernel-level vulnerabilities we studied in this dissertation. However, it might be necessary to extend the syntax to cover more vulnerabilities (e.g., those at the framework-level). Further, with the current prototype, we have to generate the policy for each vulnerability manually. Obviously, there is a gap between our previous detection results and the
policy description. As discussed in the previous chapters, the detection result was expressed as vulnerable paths, which is natural for data flow analyses, and is relatively comprehensive, so that experts can conduct further audits. However, for mitigation purposes, it is not sufficient to generate corresponding policy descriptions (semi)automatically that must be derived from some internal key values (e.g., strings) along the paths. In order to make automation possible, it may be necessary to build a more effective Intermediate Representation (IR). The IR must be able to maintain sufficient data so that any further analysis can recover necessary information easily.

The API monitor module of our system was built based on Xposed [72]. Although Xposed is a useful tool, some concerns have to be considered with respect to future work. Firstly, the current prototype does not support old devices (i.e., Android versions < 4.0.3) because of the deployment restriction in Xposed [72]. Secondly, Xposed provides more functionalities than we need, which may introduce unnecessary performance overhead. To solve these two problems, we can build a simple (but perhaps less generic) framework to serve our purposes. Further, it might be particularly helpful to address framework-level vulnerabilities in the future.

The syscall monitor module relies on the LKM to perform the hooking. Ideally, we can make it work properly if the LKM is supported by the target device. Unfortunately, that is not the case if we cannot find any existing kernel module from that device. As mentioned in Section 5.2.3, we have to modify vermagic and the CRC checksum of the module_layout to bypass the verification. We can obtain the former from the kernel image directly; however, we have not found any alternative way to obtain the latter. Of course, we can bypass the verification by using the same method as in Section 5.2.3 to bypass the device vendors’ extra restrictions. Nonetheless, it definitely weakens the security of the device and the system. As claimed before, we expect that our solution can be deployed in commodity Android devices, in which case the problem can be solved if it is a built-in kernel module.

5.5 Summary

In this chapter, we designed and implemented DVMFA, a transparent policy-based framework, to mitigate the threats introduced by app- and kernel-level vulnerabilities. DVMFA is a context-aware system that is capable of patching vulnerabilities dynamically in a fine-grained manner. Specifically, we first designed a policy language for kernel-level vulnerabilities as the base set, and then extended it to support app-level vulnerabilities with richer semantic specification and bindings. The results of our evaluation demonstrated that DVMFA was able to prevent all of the vulnerabilities we studied in Chapters 3 and 4, and it could be deployed in most of the devices we evaluated. The results also showed that the performance overhead was negligible (with less than 3% slowdown).
Chapter 6

Conclusions and Future Work

In this dissertation, we have presented a systematic approach to study and mitigate the threats of both app-level and kernel-level vulnerabilities. In particular, we first proposed SEFA, a framework that can be used to detect app-level vulnerabilities on Android devices. After that, we presented KSEFA, a framework that is capable of detecting kernel-level vulnerabilities on Android devices. SEFA and KSEFA were able to evaluate quantitatively the security impact of app- and kernel-level vulnerabilities, respectively, and the results of those evaluations demonstrated the urgent need for effective solutions. Finally, based on the results of SEFA and KSEFA, we proposed a novel and transparent, policy-based mitigation framework called DVMFA to protect vulnerable devices. DVMFA is a context-aware system that is capable of patching app- and kernel-level vulnerabilities dynamically in a fine-grained manner, and it requires no modification or recompilation of pre-loaded apps or device firmware images. The results of the evaluation showed that the vulnerabilities we studied were prevented effectively by DVMFA with negligible performance overhead.

Based on the insights we obtained in this dissertation, we propose three directions for future research on the following:

- **Detecting other types of vulnerabilities from pre-loaded apps** In this dissertation, we focused on two types of app-level vulnerabilities, i.e., those that can be used to launch permission re-delegation attacks or that lead to content leaks. In reality, there are other types of vulnerabilities. For example, the native code may be vulnerable and able to be exploited to gain advanced privileges, which, in most cases, are not necessarily root privileges, but may be with system level ones.

- **Detecting framework-level vulnerabilities** In this dissertation, we did not focus on framework-level vulnerabilities. However, this does not mean that framework-level components are immune.
It is known that framework-level vulnerabilities may be derived from the following two sources: Google’s AOSP and device vendors. Among the former, several serious vulnerabilities have been discovered in the past two years (e.g., the MasterKey bug [9]). Among the latter, our study showed that the device vendors make efforts to customize their frameworks to provide a better user experience, but this may also introduce vulnerabilities. However, to the best of our knowledge, no work has been published to date to study the framework-level vulnerabilities systematically.

- **Discovering more kernel-level vulnerabilities**  In this dissertation, our system KSEFA was capable of discovering manufacturer-specific vulnerabilities. Our system relies on existing patterns used to exploit known vulnerabilities to discover similar new ones, and it is difficult to find such patterns for kernel-generic vulnerabilities, whose exploitations may be quite sophisticated (e.g., CVE-2014-3153). However, it is indeed an interesting direction to pursue.


78


[46] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These Aren’t the Droids You’re Looking For: Retrofitting Android to Protect Data from Imperious Applications. In Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS ’11, 2011.


[49] IDC. Android and iOS Squeeze the Competition, Swelling to 96.3Smartphone Operating System Market for Both 4Q14 and CY14, According to IDC. http://www.idc.com/getdoc.jsp?containerId=prUS25450615.


APPENDIX
Appendix A

Descriptions of CVE
<table>
<thead>
<tr>
<th>CVE #</th>
<th>Source</th>
<th>Type</th>
<th>Disclosed Date</th>
<th>Community-patch Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2012-4221</td>
<td>Qualcomm</td>
<td>Integer Overflow</td>
<td>08/08/2012</td>
<td>11/15/2012</td>
<td>Integer overflow in diagchar.c in the Qualcomm Innovation Center (QuIC) Diagnostics (aka DIA) kernel-mode driver for Android 2.3 through 4.2 allows attackers to execute arbitrary code or cause a denial of service via an application that uses crafted arguments in a local diagchar_ioctl call.</td>
</tr>
<tr>
<td>CVE-2012-6422</td>
<td>Samsung</td>
<td>N/A</td>
<td>12/15/2012</td>
<td>01/24/2013</td>
<td>The kernel in Samsung Galaxy S2, Galaxy Note 2, MEIZU MX, and possibly other Android devices, when running an Exynos 4210 or 4412 processor, uses weak permissions (0666) for /dev/exynos-mem, which allows attackers to read or write arbitrary physical memory and gain privileges via a crafted application.</td>
</tr>
<tr>
<td>CVE-2013-2094</td>
<td>Linux Kernel</td>
<td>Integer Overflow</td>
<td>02/19/2013</td>
<td>04/15/2013</td>
<td>The perf_event_init function in kernel/events/core.c in the Linux kernel before 3.8.9 uses an incorrect integer data type, which allows local users to gain privileges via a crafted perf_event_open system call.</td>
</tr>
<tr>
<td>CVE-2013-4738</td>
<td>Qualcomm</td>
<td>Stack Overflow</td>
<td>07/01/2013</td>
<td>10/15/2013</td>
<td>The camera post processing engine (CPP) and video processing engine (VPE) provide an ioctl system call interface to user space clients for communication. When processing arguments passed to the VIDIOC_MSM_CPPDEQUEUE_STREAM_BUFFINFO or VIDIOC_MSM_VPEDEQUEUE_STREAM_BUFFINFO ioctl subdev handlers, a user space supplied length value is used to copy memory to a local stack buffer without proper bounds checking.</td>
</tr>
<tr>
<td>CVE-2013-6123</td>
<td>Qualcomm</td>
<td>Improper Input Validation</td>
<td>10/15/2013</td>
<td>01/10/2014</td>
<td>Multiple array index errors in drivers/media/video/msm/server/msm_camserver.c in the MSM camera driver for the Linux kernel 3.x, as used in Qualcomm Innovation Center (QuIC) Android contributions for MSM devices and other products, allow attackers to gain privileges by leveraging camera device-node access, related to the (1) msm_ctlCmd_done, (2) msm_ioctl_server, and (3) msm_server_send_cmd functions.</td>
</tr>
<tr>
<td>CVE-2013-6282</td>
<td>Linux Kernel</td>
<td>Improper Input Validation</td>
<td>10/25/2013</td>
<td>11/14/2013</td>
<td>The get哌 unp and put哌 unp API functions of the Linux kernel fail to validate the target address when being used on ARM v6k/v7 platforms. This functionality was originally implemented and controlled by the domain switching feature (CONFIG_CPU_USE_DOMAINS), which has been deprecated due to architectural changes. As a result, any kernel code using these API functions may introduce a security issue where none existed before.</td>
</tr>
<tr>
<td>CVE-2014-0196</td>
<td>Linux Kernel</td>
<td>Heap Overflow</td>
<td>12/03/2013</td>
<td>05/03/2014</td>
<td>The n Hague_write function in drivers/tty/tty.c in the Linux kernel through 3.14.3 does not properly manage tty driver access in the ‘LECHO &amp; OPPOST’ case, which allows local users to cause a denial of service (memory corruption and system crash) or gain privileges by triggering a race condition involving read and write operations with long strings.</td>
</tr>
<tr>
<td>CVE-2014-3153</td>
<td>Linux Kernel</td>
<td>Improper Input Validation</td>
<td>05/03/2014</td>
<td>06/05/2014</td>
<td>The futex_requeue function in kernel/futex.c in the Linux kernel through 3.14.5 does not ensure that calls have two different futex addresses, which allows local users to gain privileges via a crafted FUTEX_REQUEUE command that facilitates unsafe water modification.</td>
</tr>
</tbody>
</table>

Table A.1 Summary of collected vulnerabilities