UNIVERSITY OF CALIFORNIA
RIVERSIDE

From Design to Deployment: Identification and Analysis of OS Kernel Security Problems Throughout its Development Cycle

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Hang Zhang

December 2020

Dissertation Committee:

Prof. Zhiyun Qian, Chairperson
Prof. Nael Abu-Ghazaleh
Prof. Heng Yin
Prof. Chengyu Song

Acknowledgments

First and foremost, I want to thank my advisor, Prof. Zhiyun Qian, for his guidance and support during all these years. He is always willing to give me helpful and inspiring suggestions from the very high level research ideas to the low level technical details, so patient and friendly that many times he is more like a close friend to me. He has taught me so much that brings me from a novice to an independent researcher. For me, to choose him as my PhD advisor is one of the most correct decisions ever made. Perhaps more importantly, his never settled passion and curiosity for new knowledge and the brilliant creativity to achieve it show me what a great researcher and hacker should be, which will continue to guide my future life.

Many thanks to my committee members, Prof. Nael Abu-Ghazaleh, Prof. Heng Yin, and Prof. Chengyu Song, for their insightful advice and discussions on my dissertation. I have also taken their classes and learnt many useful knowledge and techniques from them. Besides, I sincerely thank Prof. Ardalan Amiri Sani from UC Irvine for his long-time support and valuable suggestions on my research direction. I also want to thank Prof. Srikanth Krishnamurthy for his support for my PhD life, and Prof. Fabio Pasqualetti from ME department for being the external committee member on my qualification exam.

I am very grateful to my lab mates and colleagues at UCR, the discussions and collaborations with them are very helpful and enjoyable, out of the lab, they also helped me a lot in my daily life. They are (randomly ordered): Zhongjie Wang, Dongdong She, Daimeng Wang, Yue Cao, Ahmad Darki, Zheng Zhang, Yizhuo Zhai, Weiteng Chen, Shitong Zhu, Yu Hao, Guoren Li, and Xiaochen Zou.

I want to say thanks to my friend, Sihuan Li, for his company and support, we have had much fun together.

Special thanks to my mentors at Microsoft Research during my internship, David Molnar, William Blum, Marina Polishchuk, and Shuvendu Lahiri, thank you for offering me this opportunity and guiding me through this unique industry experience which broadened my vision and taught me a lot.

I can never say too many “thank you” to my dear families. My parents, though far at the other side of the Pacific, keep caring and supporting me when I study abroad, their love to me is unconditional. My wife, Yao, gives me tremendous emotional support and accompany me for most of my PhD life - it should have been a long and lonely journey for me, but it is not, because of her. Thank you my families, this thesis is for you.

My research is partially supported by the following grants and funding: W911NF-13-2-0045 (ARL Cyber Security CRA), NSF Grant No.1617573, UCR Dean’s Distinguished Fellowship, UC President’s Dissertation-Year Fellowship, and other applicable grants to Prof. Zhiyun Qian. Thanks for all the providers!

Publications. This thesis includes materials from several previously published papers:

- Hang Zhang, Dongdong She, and Zhiyun Qian. “Android root and its providers: A double-edged sword.” CCS’15.
- Hang Zhang, Dongdong She, and Zhiyun Qian. “Android ion hazard: The curse of customizable memory management system.” CCS’16.
- Hang Zhang, and Zhiyun Qian. “Precise and accurate patch presence test for binaries.” USENIX Security’18.

To my families.

ABSTRACT OF THE DISSERTATION

From Design to Deployment: Identification and Analysis of OS Kernel Security Problems Throughout its Development Cycle

by

Hang Zhang

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2020
Prof. Zhiyun Qian, Chairperson

The operating system kernel security is critical for the overall system reliability, since the kernel runs in the high privileged mode and is often a part of the trusted computing base. The kernel vulnerabilities can cause severe consequences because they can be exploited by attackers to compromise many important security mechanisms (*e.g.,* Linux's access control). In this dissertation, we try to identify and study various kernel security problems across its development cycle. More specifically, we (1) Identify some kernel design and implementation flaws by systematically analyzing an Android kernel memory management sub-system, namely ION. We discover, exploit, and develop a tool to help mitigate vulnerabilities related to these flaws. (2) Analyze a class of stealthy kernel vulnerabilities: the high-order taint style vulnerabilities. We then design and implement a novel automatic static program analysis to effectively and efficiently detect such vulnerabilities in the kernel testing phase. (3) Study the attacks against the deployed kernels by analyzing multiple representative Android one-click root apps. By reverse engineering, we extract and study hundreds of well crafted kernel root exploits from these apps and alarm the community of the security risks of abusing such apps. (4) Study a security weakness in the kernel maintenance phase: the delayed or missed security patch propagation. To help battle this problem we develop a tool to accurately test the security patch presence at the binary level, which can warn the defenders of the missed security updates. Our ultimate goal is to make the kernel more secure by analyzing and fixing security issues across its whole development cycle.

Contents

List of Figures **xii**

List of Tables **xiii**

1 Introduction **1**

1.1 Background 1

1.2 Overview 2

1.3 Roadmap 5

2 Study Security Problems in Kernel Design and Implementation: Android ION Hazard **6**

2.1 Introduction 6

2.2 Background 9

2.3 Vulnerability Analysis 11

2.3.1 Problems Introduced by Unified Interface 11

2.3.2 Problems Introduced by Buffer Sharing 13

2.4 Methodology 16

2.4.1 For Unified Interface Related Issues 17

2.4.2 For Buffer Sharing Related Issues 18

2.5 Evaluation 23

2.5.1 Summary of ION related Vulnerabilities 23

2.5.2 Runtime Testing for DoS Vulnerabilities 25

2.5.3 Static Taint Analysis for Dirty Pages 27

2.6 Case Study 32

2.6.1 DoS against Heaps of Fixed Size 32

2.6.2 DoS against Heaps of Unlimited Size 33

2.6.3 DoS via Protected Memory Access 34

2.6.4 Information Leakage 35

2.7 Defense Discussion 39

2.8 Related Work 43

2.9 Conclusion 45

3 Improve Kernel Testing: Statically Detect High-Order Taint Style Vulnerabilities 46
3.1 Introduction . . . 46
3.2 Overview . . . 49
3.2.1 A Motivating Example . . . 49
3.2.2 System Architecture and Workflow . . . 50
3.3 System Design . . . 52
3.3.1 Definitions . . . 52
3.3.2 Input . . . 54
3.3.3 Static Taint Analysis . . . 55
3.3.4 Vulnerability Detection . . . 63
3.3.5 Output . . . 66
3.4 Implementation . . . 67
3.5 Limitations . . . 68
3.6 Evaluation . . . 69
3.6.1 Experiment Setup . . . 69
3.6.2 Accuracy . . . 69
3.6.3 Efficiency . . . 71
3.7 Related Work . . . 72
3.8 Conclusion . . . 74

4 Study Attacks Against Deployed Kernels: The Case of One-Click Root Apps 75
4.1 Introduction . . . 75
4.2 Publicly Available Android Root Exploits . . . 78
4.2.1 Root Exploit Impact and Coverage . . . 80
4.2.2 Exploit (Source or Binary) Availability . . . 84
4.2.3 Exploit Requirements . . . 86
4.2.4 Root Exploits Detection by Anti-Virus . . . 88
4.3 Adaptation of Root Exploits . . . 90
4.3.1 CVE-2014-3153 (Futex bug) . . . 91
4.4 Root Providers Overview . . . 93
4.5 Case Studies of Protection Mechanisms . . . 98
4.5.1 Large Root Providers . . . 99
4.5.2 Medium Root Providers . . . 103
4.5.3 Small Root Providers . . . 104
4.6 Characterization and Case Studies of Exploits . . . 105
4.6.1 Breakdown of Exploits . . . 105
4.6.2 Interaction with Advanced Security Mechanism . . . 109
4.6.3 Anti-Virus Test . . . 110
4.7 Related Work . . . 111
4.8 Discussion and Conclusion . . . 114

5 Improve Kernel Maintenance: Precise and Accurate Patch Presence Test for Binaries 116
5.1 Introduction . . . 116
5.2 Related Work . . . 119
5.3 Overview . . . 122
5.4 System Design . . . 127
5.4.1 Signature . . . 127
5.4.2 Change Site Analyzer . . . 128
5.4.3 Signature Generator . . . 132
5.4.4 Signature Matching Engine . . . 141
5.5 Implementation . . . 143
5.6 Evaluation . . . 146
5.6.1 Experiment Procedure . . . 147
5.6.2 Accuracy . . . 149
5.6.3 Performance . . . 151
5.6.4 Unported Patches . . . 151
5.6.5 Case Study . . . 153
5.7 Conclusion . . . 156

6 Conclusion and Future Work 157

Bibliography 160

List of Figures

1.1 The Kernel Development Cycle 3

2.1 ION Architecture 9
2.2 Data dependency 20
2.3 Control dependency 20
2.4 Static taint analysis tool workflow 22
2.5 Call chains of interest for CMA heap type 28
2.6 Call chains of interest for SYSTEM heap type 30
2.7 Key logic for alloc_largest_available() 31

3.1 Vulnerable Code Excerpt of CVE-2017-0608 49
3.2 System Architecture of SUTURE 51
3.3 An example of partially path-sensitive analysis 58
3.4 An example of multi-source multi-sink pairing 61

4.1 Number of exploits by layer 82
4.2 Number of exploits by year 83
4.3 Exploit requirement breakdown 88
4.4 CVE-2014-3153: kernel stack overwrite by invoking system calls 92
4.5 General Root Architecture 95

5.1 Patch of CVE-2015-8955 122
5.2 Workflow of FIBER 125
5.3 Data flow analysis of example basic blocks 134
5.4 Notation for formula (expression) annotating root instruction operands . . . 136
5.5 Illustration of the binary signature matching 139
5.6 Example Security Patches 154

List of Tables

2.1 ION heap types and instances 10
2.2 ION vulnerability summary 24
2.3 Static taint analysis result on Nexus 6P 27

3.1 Vulnerability Detectors used in SUTURE 64

4.1 Detection results of mobile antivirus 89
4.2 Root Providers List 93
4.3 Root Providers Measurement Result 96
4.4 P1's adaptation for CVE-2014-3153 (based on kernel version) 107
4.5 Anti-Virus Test Results on P1's exploits 111

5.1 Types of root instructions 135
5.2 Binary Patch Presence Test: Accuracy and Online Matching Performance . 148
5.3 Offline Phase Performance 152
5.4 Potential Security Loopholes 152

Chapter 1

Introduction

1.1 Background

The operating system (OS) plays a fundamental role in information technology since it bridges the computing hardware to its users, enabling the effective utilization of computing resources. The core component of an OS is its kernel, which on one hand directly manipulates the underlying hardware, and on the other hand exposes the management interface to the upper level applications or the end users.

There are multiple different kernels used in the real world nowadays, among which one of the most popular and versatile candidates is the Linux kernel [35]. Due to its excellent flexibility and customizability, The Linux kernel is adopted in a wide array of computing systems, from the low-power embedded systems to the high-end servers and super computers.

The prevalence of the Linux kernel makes it an appealing target for attackers, especially when considering the fact that the kernel runs in the privileged mode (*e.g.,* has the direct control for the underlying hardware) and is often a part of the trusted computing base, in other words, if the kernel is compromised, the attacker can easily gain the highest privilege of a wide range of computing systems, causing severe security consequences (*e.g.,* system malfunction, confidential data leakage, *etc.*) and financial losses. Thus, the security of the kernel is vital for the computing infrastructure of the modern world.

However, it is difficult to identify and analyze the security problems in the Linux kernel, on one hand, the kernel itself has accumulated a large and complicated code base (27.8 million LOC in 2020 [22]) after many years of open-source, community-driven development, on the other hand, since the kernel is continuously customized for and used in many different computing systems, it is no longer a solitary entity, instead, a whole ecosystem has already formed around it which contains countless forks and variants of the kernel, which may or may not be open-sourced. These factors make the Linux kernel a very unique piece of software and the kernel security an interesting yet challenging research topic, many questions can be asked regarding the security of this Linux kernel ecosystem (*e.g.,* is the kernel secure by design for all possible downstream customizations? How to effectively patch a vulnerability across the whole ecosystem?). In this dissertation, we try to answer some of these questions.

1.2 Overview

To systematically understand the challenges and opportunities in security of the kernel and its ecosystem, we need to take a holistic view of the whole process of the kernel development. Similar to other software, the kernel development also follows the common

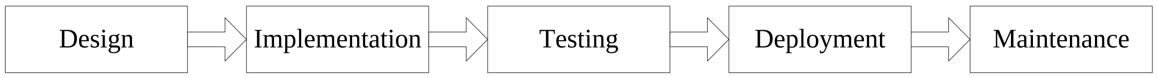


Figure 1.1: The Kernel Development Cycle

pipeline as shown in Fig. 1.1. In this dissertation, we try to raise and answer security related questions associated with each kernel development cycle, by identifying and analyzing some specific kernel security problems, more specifically:

For Design and Implementation: When designing and implementing the kernel, we want to ask the question that whether the complicated kernel and drivers are designed with security in mind, and whether there are any implementation related security problems when customizing the kernels across the ecosystem. To answer these questions, we systematically analyze an Android kernel memory management sub-system, namely ION, to identify the potential security flaws. We successfully discover and exploit severe zero-day security vulnerabilities in ION caused by both design choices and implementation defects, we also develop a tool to help mitigate some of these vulnerabilities.

For Testing: Software testing is an important step to ensure the security, a thorough and comprehensive testing can detect many hidden vulnerabilities overlooked in the design and implementation phases. Given the huge and complex kernel code base, a natural question is how we can test it more effectively and efficiently to pinpoint some stealthy vulnerabilities. For this question, we study a class of deep-hidden kernel vulnerabilities: the high-order taint style vulnerabilities, we then design and implement a novel static program analysis to automatically detect such vulnerabilities in the kernel code base, in an efficient way.

For Deployment: As mentioned before the Linux kernel can be customized and deployed in multiple different systems, and its important position makes it an attractive target for the attackers. So for the deployment phase we ask the question that how vulnerable these deployed kernels are and whether and how attackers can attack different customized kernels. To answer these questions, we study the attacks against the deployed Android kernels by analyzing multiple representative one-click root apps. By reverse engineering these apps, we manage to extract and study hundreds of well crafted kernel root exploits from them and warn the community of the security risks caused by abusing such apps.

For Maintenance: One major task in the kernel maintenance phase is to deliver security patches to the deployed kernels, in order to fix the known vulnerabilities. An important question here is whether this patching process is conducted in a timely manner (*e.g.,* so to keep the time window as small as possible for an attacker to exploit the unpatched vulnerabilities), especially when there exists a large kernel ecosystem to patch. If there are delays in the patch propagation, what can we do then to reduce them? For this question, we develop a novel static analysis tool to accurately test the security patch presence at the binary level, which helps us confirm the existence of delays during the patch propagation to even the closed-source kernel images. We believe the tool can improve the current status of kernel security since it can warn the defenders of the missed security updates on the binary level, with a high precision.

1.3 Roadmap

The remaining part of this dissertation is organized as following: in Chapter 2 we describe our study on the ION memory management system in the Android kernel to demonstrate the kernel security problems in the design and implementation phases, Chapter 3 discusses our tool to detect the high-order taint style vulnerabilities which can benefit the kernel testing, Chapter 4 illustrates our inspection and findings on the one-click root apps for the Android kernel, revealing the real-world attacks against the deployed kernels, Chapter 5 details our design and implementation of an accurate patch presence test for binaries, which can help defenders to catch the missing security updates in the kernel maintenance phase. Finally, we conclude in Chapter 6.

Chapter 2

Study Security Problems in Kernel Design and Implementation: Android ION Hazard

2.1 Introduction

Android operating system has gained tremendous popularity in the past few years thanks to the huge vendor support behind it. Unlike iOS that runs on only Apple-assembled hardware, Android is open source and encourages other vendors to build smartphones using it. This model works well as vendors do not need to build a new OS from scratch, and they can still heavily customize the phones to differentiate themselves on the market. The customization happens at all layers including hardware, OS, and applications. Major vendors such as Samsung, HTC, and Huawei all perform customizations to attract customers with features like better screens, audio, and even security [134]. While such customization itself is encouraged, it has been shown that the process of customization at the software layer often introduces security vulnerabilities [132, 134, 125].

In this study, we investigate an important OS subsystem, called ION, that is commonly customized to satisfy different requirements from the underlying hardware devices. ION [40] is a unified memory management interface widely used on ARM based Android platforms. First introduced by Google in Android 4.0, it was initially designed to replace previous fragmented interfaces originated from System-on-Chip (SoC) vendors [40]. Its main goal is to support the special requirements set by hardware devices such as the GPU and camera. For instance, some devices require physically contiguous memory to operate and some require certain cache coherency protocol for DMA to function correctly. To satisfy such requirements, on a given Android phone, ION is customized with a set of pre-configured memory heaps for the underlying hardware devices. Even though AOSP provides a set of pre-defined heap types and implementations of heap allocation and management, customization is commonplace for performance tuning and other purposes (as we will show in this chapter). In addition, for hardware devices not covered by AOSP, vendors often need to define new heap types as well as provide their own implementations of heap allocation and management.

Unfortunately, the framework for supporting such customization is not well thought out regarding its security implications. For instance, we discover that the lack of fine-grained access control to individual memory heaps can easily cause denial-of-service of specific system services or the entire OS. Moreover, its buffer sharing capability exposes different types of kernel memory to user space without being screened carefully for security consequences. To demonstrate the seriousness of the identified vulnerabilities, attack demos and analysis can be found on our project website [1]. In this chapter, we make three main contributions:

- We systematically analyze the security properties from the design and implementation of ION, and reveal two major security flaws that lead to many vulnerabilitis and corresponding exploits, which are already reported to and confirmed by Google.
- To detect specific vulnerability instances, we develop both a runtime testing procedure and a novel static taint analysis tool that help uncover vulnerabilities on newest flagship models like Nexus 6P and Samsung S7 running Android 6.0 and 7.0 preview (latest at the time of writing).
- By analyzing the root causes of the problem, we propose alternative designs that preserve the ION functionality while eliminating its security flaws. We believe the lessons learned can shed light on future designs of customizable and extensible memory management system.

The remaining part of this chapter will be organized as following: §2.2 will briefly introduce some ION-related background knowledge, §2.3 will give a thorough analysis of ION related vulnerabilities, §2.4 will detail our methodology to systematically identify the vulnerabilities on a wide range of Android devices, §2.5 will summarize the vulnerabilities we find on various devices and evaluate the effectiveness of our methodology, §2.6 will demonstrate our actual exploitations against ION related vulnerabilities on some representative devices. In §2.7, we discuss possible defense against the vulnerabilities. §2.8 discusses the related works and §2.9 will conclude the chapter.

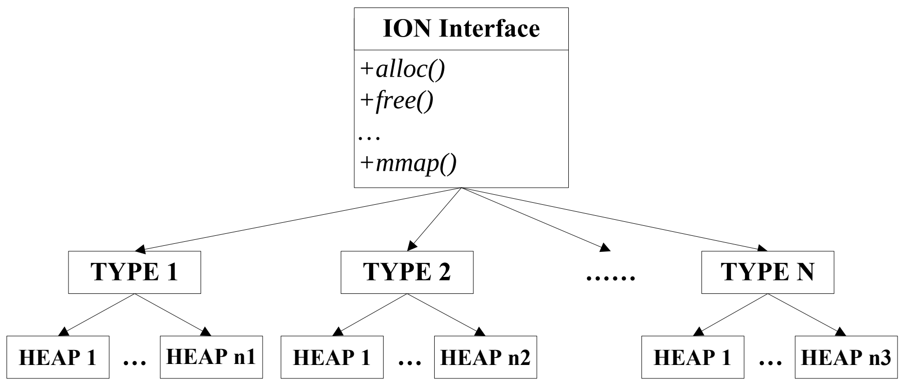


Figure 2.1: ION Architecture

2.2 Background

As briefly described, ION is designed to achieve two main goals. First, it aims to support hardware devices with diverse memory requirements. Prior to ION, different SoC vendors achieve this through proprietary and mutually incompatible interfaces such as PMEM for Qualcomm, NVMAP for Nvidia, and CMEM for TI [40]. System and application developers have to customize their code heavily for all such interfaces to ensure that the code can work across all different platforms. This problem is greatly alleviated since the introduction of ION that defines a common interface irrespective of SoC manufacturers. The underlying implementation in the form of a driver can be customized by SoC and smartphone vendors to guarantee that they return the correct type of memory asked by the user space.

As with most interfaces exposed to user space, the unified ION interface is exposed through the */dev/ion* file, which can be manipulated through `open()` and `ioctl()` system calls. The specific set of supported operations include "alloc" and "free". The user space

Types	Instance	Page source	Contig*
SYSTEM	Nexus 6P:system	alloc_pages() of buddy allocator	n
SYSTEM_CONTIG	Nexus 6P:kmalloc	alloc_pages() of buddy allocator or kmalloc()	y
CARVEOUT	Samsung S7:camera	preserved memory region, not reusable	y
CMA	Nexus 6P:qsecom	preserved memory region, reusable	y
SECURE_CMA	Nexus 6P:mm	preserved memory region, reusable	y
CHUNK	N/A	preserved memory region, not reusable	y
REMOVED	N/A	preserved memory region, not reusable	y
EXYNOS	Samsung S4: exynos_noncontig_heap	alloc_pages() of buddy allocator	n
EXYNOS_CONTIG	Samsung S4: exynos_contig_heap	preserved memory region, reusable	y
CPUDRAW	Huawei Mate8: cpudraw_heap	preserved memory region, not reusable	y

[1] N/A means that we have not observed actual instances of the heap type
[2] Even the same type can have different implementations on different devices
* Whether the allocated memory region is physically contiguous or not

Table 2.1: ION heap types and instances

code needs to specify a heap id from which the memory should be allocated. As shown in Figure 2.1, each ION heap has an assigned name, id, and more importantly, an associated heap type that is pre-defined for a particular Android device. Table 2.1 illustrates the set of AOSP-defined heap types, along with a selected subset of customized heap types we encounter in the studied Android devices. Even though incomplete, it illustrates the complexity of ION with heaps of different types and properties. Some heap types may appear to have similar properties: CMA and SECURE_CMA. However, they actually serve different purposes. CMA is accessible by third-party apps and system services. However, SECURE_CMA is usually intended to be used by trusted world (See TrustZone [6]), thus inaccessible from user space. Also, we omit another dimension, cache coherency, which is not the focus of this study.

Generally the heaps fall into two categories: 1) Unreserved. The most representative one is the SYSTEM heap, whose memory provider is the low-level buddy allocator according to our analysis , the same as memory allocated through `malloc()`. 2) Reserved. This includes CARVEOUT and CMA heaps that involve memory set aside at boot time so as to combat memory fragmentation at runtime [101].

The second goal of ION is to allow efficient sharing of memory between user space, kernel space, and the hardware devices. This is achieved by sharing memory pages directly to avoid copying. Specifically, following the ION interface explained earlier, once memory is allocated successfully from a heap, a file descriptor is returned to user space which can be subsequently used to invoke `mmap()` to map the allocated pages into user space. This feature can be handy in many scenarios. For instance, in the case where both software and hardware rendering are needed for graphics processing, libraries such as OpenGL can manipulate the memory in user space easily and a GPU can also populate the same piece of physical memory with zero copying.

2.3 Vulnerability Analysis

So far, we have explained the design philosophy of ION including 1) unified memory management interface for ease of use and 2) memory sharing support between user and kernel space. Interestingly, each one introduces a new class of vulnerabilities. In this section, we will unveil the root causes of the security flaws.

2.3.1 Problems Introduced by Unified Interface

As mentioned in §2.2, ION uses a unified interface */dev/ion* for all types and instances of memory heaps it manages. Unlike the general memory allocated through `malloc()` in user space, ION heaps come with different sizes and purposes, which require a different security design than the one for general memory. In Android, an application can allocate "unlimited" amount of general memory through `malloc()`. Because it is general memory, applications may have legitimate reasons to allocate and use a large amount of memory (*e.g.,* 3D gaming apps). The only time when memory allocation fails is when the system is running out of memory.

Unfortunately, ION inherited the security design for general memory. There does not exist any limit on how much memory one can consume in ION heaps, causing potential DoS attacks. Even worse, due to the fact that third-party apps have legitimate reasons to allocate memory from at least one heap type (for graphic buffers), the unified */dev/ion* interface needs to have a relaxed permission that allows anyone in user space to have access to potentially *all* ION heaps. Indeed, on all 17 phones we studied, the file permission of */dev/ion* is always world-readable [1]. There exists no other security mechanisms (*e.g.,* access control) beyond the file permission and therefore any app can allocate any amount of memory from any ION heaps (but no more than the max available of a certain heap). Due to characteristics of different heap types, such a capability can lead to two different DoS attacks:

For fixed size heaps. Certain ION heap types such as CARVEOUT and CMA have a pre-determined size and region from which users can allocate memory from. These heaps are typically used for various system functionalities, *e.g.,* "audio" heap is used by "mediaserver" on Nexus 6P to perform audio playback. As mentioned in §2.1, during the Android device customization process, the available heap types and instances are tailored to satisfy the

[1]Readable on */dev/ion* in fact allows both memory allocation and memory mapping to user space [40].

need of hardware devices. In the above example, the "audio" heap is designed to work with a specific audio chip. As long as a user exhausts all free space of a certain heap, related system functionalities will stop working due to the failure to get required memory from the specific heap, *e.g.,* sound/music playback on Nexus 6P will be disabled if "audio" heap is occupied by a malicious app. In some cases, critical service failure can even cause the whole system to crash as shown in §2.6.

For unlimited size heaps. Some heap types, such as SYSTEM, have no pre-reserved memory regions. According to our analysis, memory allocated from the SYSTEM heap is not correctly accounted for as part of the memory usage of the calling process. Thus, from SYSTEM heap a process can request as much memory as the current system can supply. When a user process drains too much memory from such heaps, the performance of the whole system will be affected. Besides, due to the existence of Android low memory killer [2], other innocent processes may get killed to release more memory in such a situation. More detailed analysis is given in §2.6.

One may argue that the above problems can be solved by integrating a general access control or quota limitation mechanism into current ION interface, however, this may not be an easy solution as will be discussed in §2.7.

2.3.2 Problems Introduced by Buffer Sharing

As mentioned earlier, the zero-copy buffer sharing among user space, kernel space, and hardware devices is one of the main goals of ION. While some specific heap types such as SECURE_CMA may deny any access from user space, there is no general restriction in the

ION framework; therefore users can allocate memory from most heaps and map them into user space for read and write operations. This can lead to two different security problems:

System crash due to hardware protection. Some heaps' memory regions can be protected by hardware security mechanisms like TrustZone [6] so that any access from untrusted world will cause a protection exception which usually leads to a system crash or reboot. Unfortunately ION buffer sharing unexpectedly makes such protected heaps accessible to untrusted world apps, as will be shown in §2.6.

Sensitive information leakage. ION memory are drawn from various memory heaps, whose allocation functions correspond to low-level kernel functions such as `kmalloc()` and `dma_alloc_*()`. By default, many of them do not zero the newly allocated pages for performance reasons. The original assumption was that such pages will never get mapped to user space directly and hence safe to use in kernel. Unfortunately it is no longer correct with the introduction of ION that exports such heaps to user space for buffer sharing. We confirm that it is far from rare cases that buffers allocated from ION heaps contain dirty pages. A large number of Android devices, including the newest models like Nexus 6P, have one or more ION heaps failing to clear allocated buffers before handing them to user space, which eventually enables any third-party app to access sensitive information leaked from kernel, system services, and user applications.

2.3.2.1 Root Cause Analysis of Unzeroed Pages

After a careful investigation, we summarize two main reasons for unzeroed pages:

Customization. ION by design has good extendability and supports customization, as described in §2.1. Different vendors can have their own choices about which heap types to use and how they will be implemented. They can also implement new heap types by themselves as shown in Table 2.1. Thus, even though all the default heap types have zeroed their allocated buffers in AOSP common branch kernels since 3.10, in practice, almost all Android devices are shipped with customized ION implementation which do have the dirty page problem.

Kernel memory allocation functions with complicated behaviours regarding buffer zeroing. Unlike the relatively simple and limited interfaces for user-space memory allocation like `malloc()`, there exist many different memory allocation interfaces in kernel space, which will be directly used by various ION heap types. `malloc()` typically involves the system call `brk()` and `mmap()` where virtual pages are returned first. Upon accessing such pages by the program, a page fault occurs which triggers the OS to locate a physical page and map the accessed virtual page to it. Due to the obvious security risks, the OS always zeroes the physical page before mapping it to user space (unless it is a page recycled from the same process).

In contrast, other kernel memory allocation functions are diverse, largely undocumented, and not well understood. They generally fall into three categories:

(1) *Guaranteed zeroing.* Interfaces like `kzalloc()` are guaranteed to zero the allocated memory, which do not pose any threats even when exposed to user space through ION.

(2) *Expected to zero but actually may not.* Some functions will decide whether to zero the memory based on some function parameters like *GFP_ZERO* flag used in `alloc_pages()` of buddy allocator. However, there is no guarantee that the zero operation will be performed. In our analysis, we found that some functions, like `arm64_swiotlb_alloc_coherent`, do

accept a parameter deciding whether to zero the allocated buffer, but the function implementation simply does not honor such a parameter at all. A similar issue was reported previously in [41].

(3) *Undecidable and undocumented zeroing behaviour.* There also exist other functions where it is not obvious whether the returned pages will be zeroed. `gen_pool_alloc_aligned` is one such function that is usually used by CARVEOUT heaps.

The confusing behaviors of various kernel memory allocation functions makes it difficult for developers to decide whether they should zero the buffer after invoking any kernel memory allocation function. On one hand, failure to zero the buffer may cause information leakage, while on the other hand, repeated zeroing operations may affect the overall performance, especially on embedded platforms. As an example, some buffer zeroing logic is surprisingly commented out intentionally for several ION heap types according to the kernel source code for Huawei Mate 8, a popular device running Android 6.0. We suspect that developers are trying to avoid the extra performance penalty, but the end result is a severe security flaw as will be demonstrated in §2.6.

2.4 Methodology

In this section, we will present our methodology to systematically test and discover the vulnerabilities uncovered in §2.3.

2.4.1 For Unified Interface Related Issues

The goal is to test whether a third-party app can indeed occupy memory from different heaps entirely to cause DoS attacks. Further, we want to understand what specific system functionalities can be targeted using which heaps. To this end, we design a simple runtime testing procedure as follows: given an Android device, we first enumerate all available ION heaps (declared through the Device Tree file [14]), identify their type and size information; we then try to allocate buffers from them with an ION memory probing app we develop. Once we find a heap able to provide memory to our app, we will further try to exhaust all available memory resources remained in the heap. This can be automatically done by our probe program, which will try to allocate a buffer as large as possible in each iteration of a loop and terminate the loop if no more memory can be allocated. The largest available buffer size in each iteration is decided by an efficient binary-search style probing. For unlimited sized SYSTEM heaps, we will also try to allocate as much memory as we can, until exceptions occur, such as our process getting killed by low memory killer.

As soon as an ION heap is exhausted, we will monitor system behaviors to see whether there will be any anomalies. Usually the heap name will give a good indication about what system behaviors to watch, for example, the name "audio" suggests that the heap should be used for audio data processing, then we will focus on the issues such as whether the system can still play sound normally. For SYSTEM heaps, we mainly focus on questions like whether the system performance will be affected or whether there are other processes get killed by low memory killer. If kernel and platform source code for the target device is available, we will also try to take a reference of it to figure out how the dedicated

ION heaps will be used, which can help us find the potential DoS problems more efficiently and precisely.

2.4.2 For Buffer Sharing Related Issues

If a heap not only allows our app to allocate memory from it but also enables it to access the allocated buffers, then we will first attempt to access the buffers. In some cases, a simple memory read operation can already cause a system crash as described in §2.3. If the buffer can be accessed without causing exceptions, we will then determine whether the buffers from the current heap may contain dirty pages. This can be done in two ways:

Blackbox testing. We can simply exhaust the free space of a certain heap and read the allocated buffers to see whether they contain any non-zero bytes. To avoid the cases where the heap may not be populated by other services yet, *e.g.,* camera has not been used yet and therefore no data has been stored in the heap, we could write to the heap first and later on read from it again from another app to see if the data remain. The challenge with such a blackbox testing approach is that the behaviors can be dependent on the system state and the parameters we pass through the ION interface, which may not be easy to determine; this can lead to inaccurate assessments. In addition, blackbox testing also requires access to actual devices.

Program analysis. Alternatively, if the kernel source code for an Android device is available, which usually is the case due to open source licensing requirements, we can in fact accurately determine this via static analysis on the source code. As discussed before in §2.3.2, since the behaviors of kernel memory allocation functions are complex and in many cases not well documented, program analysis can automate the process and greatly reduce the manual effort.

2.4.2.1 Static Taint Analysis on Buffer Zeroing

To fulfill this task, we design and implement a novel *static taint analysis* tool to analyze the zeroing behaviors of memory allocation functions. Our design is based on three key observations:

(1) Most, if not all, memory allocation functions will take a parameter indicating the size of the requested memory. We consider such "size" parameters as *taint source*.

(2) Usually the zeroing operations will be performed through some common utility functions such as `memset()`, which will be considered the *taint sink*.

(3) When buffer zeroing operations occur in memory allocation functions, the amount of memory zeroed should be dependent on the "size" taint source. In other words, the taint source should propagate to the sink to indicate a true buffer zero operation.

The lack of an information flow from the source ("size") to sink (functions such as `memset()`) in an ION heap allocation function indicates that the function does not zero the buffer before returning it to user space. It is worth noting that the analyzed memory allocation functions may include zeroing operations for some internal or temporary data structures other than the allocated buffer, in which case may cause confusion. Our intuition is that such data structures will not be dependent on the "size" taint source and therefore can be eliminated automatically.

```
allocate_1(…,size,...){
  order=log_2(size);
  some allocation operations;
  memset(addr,0,2^order);
}
```

Figure 2.2: Data dependency

```
allocate_2(…,size,...){
  some allocation operations;
   for(i=0;i<size;i+=4096)
      memset(addr+i,0,4096);
}
```

Figure 2.3: Control dependency

Design Considerations. Even though the formulated problem is clearly defined, there are still several complications that need to be carefully considered. First, taint propagation typically has two forms: data dependency (explicit flow) and control dependency (implicit flow). We need to decide whether to track data dependency alone or both. Most static taint analysis tools focus on only data dependency [98, 45, 72, 124]. However, in the case of memory zeroing operations, the decision may not be so straightforward. We illustrate two real world examples (simplified) we encounter in Figure 2.2 and Figure 2.3. `allocate_1()` round up the requested size to the nearest power of 2 before allocating a buffer and `memset()`ing it. The "order" variable is data dependent on the "size" taint source; therefore, it is sufficient to consider data dependency only in this case. In contrast, `allocate_2()` decides to invoke `memset()` to zero the allocated buffer page by page. No data dependency exists from "size" to the parameter of `memset()`. Instead, a tainted control dependency exists from "size" to `memset()` as the loop condition is dependent on "size". In this case, we will need to follow all function calls after meeting a tainted control dependency so as to not miss any sink functions (*e.g.,* `memset()`). However, such strategies can incur false positives, as we will show in §2.5.3. We acknowledge that it is an inherently difficult problem to propagate taint through control dependencies, as is recognized in prior work [52]. As an alternative solution, manual intervention can be used to determine the propagation rules upon each tainted control dependency. We give a complete walkthrough of the methodology in §2.5.3.

Second, it is possible that the ION memory allocation function may internally invoke different low-level kernel memory allocation functions (*e.g.,* fall back to a different function if a previous one fails). Therefore, even if there exists an information flow from source to sink (for certain program paths), it does not rule out the possibility that another program path does not zero the buffer. To address this issue, our tool will output the unique call chains associated with the taint paths and guide the developer to look for other low-level memory allocation functions; our assumption here is that there must exist a different memory allocation function for each type of low-level memory allocator. Depending on the system state, or the result of an earlier memory allocator, ION may choose to invoke a different memory allocator (again, in the form of a separate function). With this assumption, the tool can output the callees for each memory allocation function in the tainted call chain. If developers recognize any callee that also appear to be a memory allocation function (and takes in a tainted argument), they can query the taint analysis result to see if the callee has encountered any zero operation down the line. If so, the tool simply repeats the same procedure to look for additional candidate callees. Otherwise, we conclude that there does exist a program path that performs memory allocation without buffer zeroing. In §2.5.3, we use the SYSTEM heap on Nexus 6P as a case study to explain the methodology in detail.

Third, theoretically buffer zeroing can also occur during memory release functions, *e.g.,* `free()`. In practice, we find that ION heaps always have relatively simple logic in memory release functions and they almost never zero buffers in them. This could be due

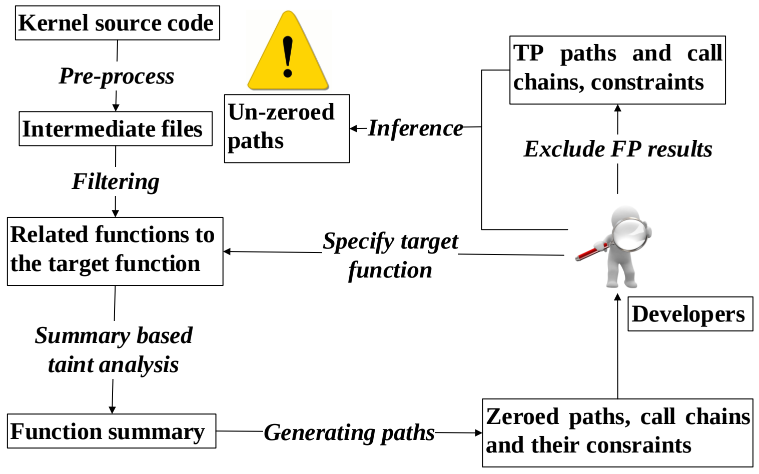


Figure 2.4: Static taint analysis tool workflow

to the fact that memory allocation functions can opportunistically skip zeroing operations if the pages are from the same process, *e.g.,* memory allocated through `malloc()`. Also, if the released memory is not be reused afterwards, the zeroing is simply wasted. Therefore, in our analysis, we focus on analyzing memory allocation functions, which are orders of magnitude more complex and may or may not contain zeroing operations.

Implementation. We implement the static taint analysis based on STAC [33], an open source static taint analysis tool. The workflow is described in Figure 2.4. Given the kernel source code of a specific Android device, we first perform pre-processing using GCC to produce .i files with expanded macros and include files. Then we perform filtering to exclude the functions that are never reachable from the ION functions. Finally, we implement a flow-sensitive taint analysis engine that takes in the entry function, *i.e.,* ION memory allocation functions, as well as the source and sink. When the taint engine finishes the computation, we output the taint paths where the source "size" can successfully propagate to sinks (*e.g.,* `memset()`). Finally, we group the taint paths into call chains for developers to inspect and confirm. Based on the true positives, developers can then follow the procedure described above to locate unzeroed paths.

2.5 Evaluation

In this section, we will first give the experiment setting and a summary of discovered vulnerability instances. Then we will evaluate the effectiveness of the methodology to discover two classes of vulnerabilities, including how successful is the static taint analysis tool in practice. Finally, we will use case studies to highlights important findings.

Scope. We have analyzed 17 Android devices in total, which are listed in Table 2.2. They cover a wide range of devices such as Nexus, Samsung, to HTC. All of them are verified through runtime testing and source code analysis. Out of 17 devices, we have experimented in detail using 8 devices, for which we have constructed exploits to confirm the existence of vulnerabilities.

2.5.1 Summary of ION related Vulnerabilities

After applying the methodology described in §2.4, we report our findings in Table 2.2. Note that we group the tested devices based on their hardware platforms (SoC vendor and model) and kernel versions. This is because hardware devices are the most common reason for customization of ION. Generally speaking, devices sharing the same hardware platform will have similar configurations regarding ION heap types and instances. In addition, different phone vendors and kernel versions may also have an impact.

Devices	Platform	Kernel	A1		A2		A3		A4	
			effect	heap	effect	heap	effect	heap	effect	heap
Galaxy S7*	exynos5	3.18.14	D1	T3:N7,N16 T4:N15,N11 T5:N17,N14, N12,N13	yes	T1:N18	no	no	L1	T3:N7,N16
Galaxy S6*	exynos5	3.10.61	D1,D4,D3	T3:N5,N3	yes	T1:N18	no	no	L1	T3:N3,N5
Meizu Pro 5	exynos5	3.10.61	D1,D4	T3:N5,N3	yes	T1:N18	no	no	L1	T3:N3,N5
Nexus 6P*	msm8994	3.10.73	D1,D2	T4:N2,N4,N1	yes	T1:N19 T2:N20	yes	T4:N6	L2,L4	T4:N1,N4,N2
LG V10	msm8992	3.10.49	D1,D2	T4:N2,N4,N1	yes	T1:N19 T2:N20	no	no	L2,L4	T4:N1,N4,N2
HTC A9	msm8952	3.10.73	D1,D2	T4:N2,N4,N1	yes	T1:N19 T2:N20	yes	T4:N6	L2,L4	T4:N1,N4,N2
Oppo R7sm*	msm8916	3.10.49	D2	T4:N2,N4,N1	yes	T1:N19 T2:N20	yes	T4:N6	L2,L4	T4:N1,N4,N2
Nexus 5X*	msm8992	3.10.73	D1,D2	T4:N2,N4,N1	yes	T1:N19 T2:N20	no	no	L2,L4	T4:N1,N4,N2
Xiaomi 4C*	msm8992	3.10.49	D2	T4:N2,N4,N1	yes	T1:N19 T2:N20	no	no	L2,L4	T4:N1,N4,N2
vivo Y927	msm8916	3.10.28	D2	T4:N2,N4,N1	yes	T1:N19 T2:N20	no	no	L2,L4	T4:N1,N4,N2
Nexus 5*	msm8974	3.4.0	D2	T4:N2,N4 T3:N1,N6	yes	T1:N21 T2:N20	no	no	L2	T3:N1
LG D950*	msm8974	3.4.0	D2	T4:N2,N4 T3:N1,N6	yes	T1:N19 T2:N20	no	no	L2	T3:N1
HTC D816	msm8226	3.4.0	D2	T4:N2,N4 T3:N1,N6	yes	T1:N19 T2:N20	no	no	L2	T3:N1
Oneplus One	msm8974	3.4.0	D2	T4:N2,N4 T3:N1,N6	yes	T1:N19 T2:N20	no	no	L2	T3:N1
Galaxy note 3	msm8974	3.4.0	D2,D4	T4:N2,N4 T3:N1,N6,N7	yes	T1:N19 T2:N20	no	no	L1,L2	T3:N1,N7
Huawei P9	hi3650	3.4.90	D5	T3:N8	yes	T1:N22 T2:N10	no	no	L4	T4:N9 T2:N10
Huawei Mate8*	hi3650	3.4.86	D5	T3:N8	yes	T1:N22 T2:N10	no	no	L4	T4:N9 T2:N10

* Devices with detailed experimentation and constructed exploits
- "effect" column shows only attack effects known to us (which can be incomplete)
[A] **Vulnerabilities Classification:** **A1**:DoS for limited size heaps **A2**:DoS for unlimited size heaps **A3**:System crash due to protection exception **A4**:Information leakage
[D] **DoS Attack:** **D1**:fingerprint **D2**:audio **D3**:video **D4**:camera **D5**:system crash
[L] **Information Leakage:** **L1**:camera **L2**:audio **L3**:video **L4**:general user apps
[T] **Heap Type:** **T1**:SYSTEM **T2**:SYSTEM_CONTIG **T3**:CARVEOUT **T4**:CMA **T5**:SECURE_CMA
[N] **Heap Name:** **N1**:audio **N2**:adsp **N3**:video **N4**:qsecom **N5**:secdma **N6**:pil **N7**:camera **N8**:carveout-heap **N9**:ion-dma-heap **N10**:system-contig-heap **N11**:video_nfw **N12**:video_fw **N13**:video_scaler **N14**:video_frame **N15**:crypto **N16**:gpu_crc **N17**:gpu_buffer **N18**:ion_noncontig_heap **N19**:system **N20**:kmalloc **N21**:vmalloc **N22**:system_heap

Table 2.2: ION vulnerability summary

In our study, we focus on three main general hardware platforms: MSM (Qualcomm), Exynos (Samsung), and Hisi (Huawei). Each platform can also include different models, *e.g.,* Snapdragon 810 and 820 correspond to two different Qualcomm SoCs, along with numerous Android and kernel versions. As we can see in the Table, there exist a variety of vulnerable ION heaps (up to 22 instances across all devices).

In the Table, we breakdown the vulnerabilities into 4 categories, along with their corresponding attack effects and vulnerable heaps (types and instances). For instance, regarding A1: DoS against heaps of limited size, all experimented devices are vulnerable in one form or another. On Huawei devices specifically, A1 attack can even cause the whole system to crash directly. Regarding A4: information leakage, all studied Android devices have unzeroed memory that can be leaked from different heap types. The most surprising result is that 9 out of 17 devices have the information leakage vulnerabilities that allow a malicious app to obtain dirty pages used by other apps, which can contain sensitive information such as passwords, credit card numbers, or even secret keys.

Although the number of Android devices we analyze is limited, they do cover most representative manufactures, hardware platforms and software versions, thus we can infer that most Android devices to date are affected by ION-related vulnerabilities. Specifically, Nexus 6P, Samsung Galaxy S7, and Huawei Mate 8 represent the latest devices from each manufacturer, all of which have both DoS and information leakage vulnerabilities.

2.5.2 Runtime Testing for DoS Vulnerabilities

The runtime testing procedure described in §2.4.1 is overall effective for most devices; however, when applying this methodology, we did encounter some special cases in which the normal routine fails to give useful results even though we can successfully allocate arbitrary memory buffers from a certain heap. We describe them below.

Failure to identify any DoS vulnerabilities. For certain heaps with limited size on some devices, even after we occupy all of its free space, no issues can be observed. After looking into these cases, we conclude two main reasons for this: 1) Some heaps will be rarely, if not never, used by their host devices. For instance, we cannot observe any utilization of "kmalloc" heap, whose type is SYSTEM_CONTIG, on Nexus 6P in our experiments. Besides, there exists other heaps that may be used in only the early stage of system booting, such as "pil" heaps on some devices that are used to load certain firmware images during the boot process.

Vulnerabilities depending on proper timing. In some cases, we can successfully perform DoS attacks against certain system functionalities by exhausting specific heaps, but not at arbitrary points in time. In the case of Samsung S6's fingerprint authentication service, a CARVEOUT heap named "secdma" is used to fulfill its task. If a malicious app occupies the entire heap ahead of time, then fingerprint service will stop functioning. The challenge is that the service itself typically occupies the heap when the screen is locked and releases it only when the screen is unlocked. Generally speaking, our testing methodology may not always be able to catch the correct timing; however, such vulnerabilities do exist and are simply harder to trigger. Manual investigations are performed to catch these cases as reported later in §2.4.1.

Heap Type	TP*	FP*	FN*	Un-zeroed paths?	Analysis/Actual result	Involved allocation function
SYSTEM	1158(8)	5	0	n	zeroed/zeroed	alloc_pages()**
SYSTEM_CONTIG	288(6)	0	0	n	zeroed/zeroed	alloc_pages()**
CMA	4(4)	2	0	y	un-zeroed/un-zeroed	dma_alloc_attrs()
CARVEOUT	0(0)	0	0	y	un-zeroed/un-zeroed	gen_pool_alloc_aligned()

* TP, FP, and FN refer to the amount of call chains. The number in brackets indicate the number of paths.
** The parameters passed to alloc_pages() are different for the two heap types.

Table 2.3: Static taint analysis result on Nexus 6P

2.5.3 Static Taint Analysis for Dirty Pages

Next, we evaluate the effectiveness of the static taint analysis described in §2.4.2.1, using Nexus 6P as a case study. The source code we analyze is from kernel version 3.10.73. In total, it takes about 9.5 hours for the tool to analyze the memory allocation functions for 4 ION heap types — SYSTEM, SYSTEM_CONTIG, CMA, and CARVEOUT — on a server with Intel Xeon E5-2640 V2 CPU and 64GB physical memory. The analyzed LOC is over 10,000. We omit the remaining heap types that fall in two categories: 1) the ones that never have any instances, *e.g.,* REMOVED type; 2) the ones that deny any `mmap()` requests from user space, *e.g.,* SECURE_CMA, indicating that it is impossible to access the memory although they may contain dirty pages.

We summarize the result of the tool on Nexus 6P's kernel source code in Table 2.3. We output the number of taint paths as well as the corresponding unique call chains (the number in brackets indicates the number of unique call chains). We confirm the true positives, false positives, and false negatives by manually analyzing the source code. The false positives are relatively easy to deal with, as a developer or researcher can quickly inspect the output path (or even the function call chain) to confirm them (we will explain

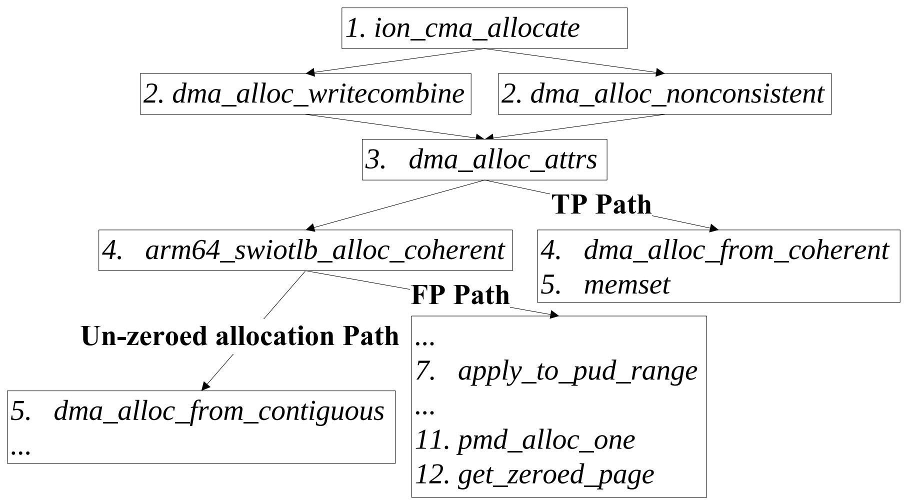

1. ion_cma_allocate
2. dma_alloc_writecombine
2. dma_alloc_nonconsistent
3. dma_alloc_attrs
TP Path
4. arm64_swiotlb_alloc_coherent
4. dma_alloc_from_coherent
5. memset
FP Path
Un-zeroed allocation Path
...
7. apply_to_pud_range
...
11. pmd_alloc_one
12. get_zeroed_page
5. dma_alloc_from_contiguous
...

Figure 2.5: Call chains of interest for CMA heap type

the FP cases of CMA heap later). For false negatives, it is more problematic as we may not even realize this and incorrectly report that there are no zeroing operations while in fact there are. One potential source of false negatives is the incomplete set of sink functions considered, which include the common functions such as `memset()` and `bzero()` so far. However in our evaluation, we found that `memset()` and its wrappers are the only used sinks.

CMA heap type. We first discuss the false positive taint paths in CMA heap. We group such paths into call chains as shown in Figure 2.5 (labelled “FP path”). When we look at the results, the taint analysis in fact correctly outputs the taint result according to the source and sink definition. Unfortunately, in some cases, the allocation of auxiliary data structures, such as page table entries, is also dependent on the size of requested buffer. Specifically, the auxiliary data here is the page table entries that are created and subsequently zeroed. In other words, the source (“size” parameter) indeed propagated to the sink (“zeroing” operation); the zeroing operation is simply not applied to the returned buffer. Fortunately, by simply looking at the function names involved in the FP call chain in Figure 2.5, it is easy to conclude that this call chain is for allocating pages to hold page table entries given the term “pmd” (page middle directory).

According to our evaluation, there do exist true positive paths that result in zeroed buffers. The cases are also shown in Figure 2.5. Now the question is whether there still exist unzeroed paths other than the zeroed path. As stated previously in §2.4.2.1, we walk backwards along the true positive call chains and look for branches that may invoke other memory allocation functions that do not belong to any taint call chain. As shown in Figure 2.5, starting from `dma_alloc_from_coherent()`, we walk backwards to its caller `dma_alloc_attrs()` and enumerate all of its callees. By looking at the names of the callees, we locate `arm64_swiotlb_alloc_coherent()`, which also takes in a tainted parameter “size”, and appears to be a memory allocation function. We then cross reference the function name with our taint analysis results. In this case, we found no taint call chains that involve `arm64_swiotlb_alloc_coherent()`, which indicates the possibility of a path where the allocated memory is unzeroed. Upon reading the code, we realize that the code tries to allocate memory through `dma_alloc_from_coherent()` first, and fall back to `arm64_swiotlb_alloc_coherent()` only when an error is returned earlier. During our runtime testing on Nexus 6P, `dma_alloc_from_coherent()` appears to be failing all the time and therefore we are able to successfully obtain dirty pages in CMA heap.

SYSTEM heap type. SYSTEM heap is complex and involves many paths leading to zero operation, a simplified call graph is shown in Figure 2.6. All 5 FP call chains are introduced

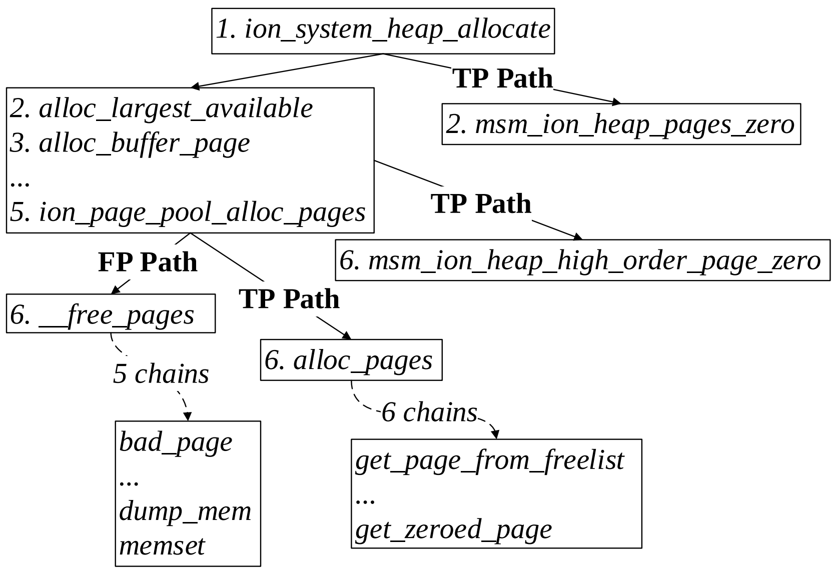


Figure 2.6: Call chains of interest for SYSTEM heap type

by the function `__free_pages()`, from which an error branch is eventually triggered which invokes `memset()`. When we look at the issue closely, it all started from the function `alloc_largest_available()`, whose key logic is depicted in Figure 2.7. Upon close inspection, there exists a control dependency (implicit flow) that caused the problem. The function essentially runs in a loop to identify the closest round-up of the request memory size in the power of 2, and use the round-up value to allocate memory. Here the constant array $order[i]$ simply pre-defines all possible round-up values. Note that the $size$ parameter (taint source) is compared against the round-up value $orders[i]$, which results in a control dependency. The variable $order[i]$ itself is not tainted as it is a read-only constant. When it is passed over to `alloc_buffer_page()`, we lose track of the taint. In reality, $order[i]$ is semantically derived from the taint source $size$ (round-up of size), however, traditional taint propagation rules are unable to catch this case. Therefore, we have to record all function

```
struct page_info *alloc_largest_available(...,size,...){
    for (i = 0; i < num_orders; i++) {
            if (size < order_to_size(orders[i]))
                    continue;
            page = alloc_buffer_page(...,orders[i],...);
    }
}
```

Figure 2.7: Key logic for alloc_largest_available()

invocations after the control dependency and report whenever a sink (*e.g.,* `memset()`) is encountered, regardless of whether its parameters are tainted (we had to implicitly assume that all parameters of the sink is tainted).

Of course, in practice, such a coarse-grained control taint propagation rule is likely going to introduce false positives. It is inherently a challenge to deal with control taint propagation, as widely acknowledged in previous studies [52]. As an alternative solution, a developer can resolve the control taint manually. In this case, since we know $orders[i]$ is essentially derived from the taint source $size$, one can simply taint $orders[i]$ directly to avoid the false positives.

Aside from the false positives, we wish to point out an interesting observation. Figure 2.6 appears to suggest that there are multiple paths to zero the allocated buffer and there exists only one path that actually allocates memory. Specifically, both `msm_ion_heap_pages_zero()` and `msm_ion_heap_high_order_page_zero()` are simply zeroing a buffer without allocating any memory. Only `alloc_pages()` is allocating memory (as well as zeroing the buffer afterwards). Therefore, one may think whether the buffer is zeroed for more than one time. However, upon a closer look, we realize that two of the sinks are not really performing the zeroing operations. Both `msm_ion_heap_high_order_page_zero()` and `alloc_pages()` depend on a flag "GFP_ZERO". Only if it is set will they zero the buffer. In this particular case, the flag is not set for either function to avoid repeated zeroing that can waste CPU cycles. It is interesting to see how complex the memory allocation can be and how hard the developers need to try to ensure security as well as performance. This once again shows the benefit of a program analysis tool to help developers make correct implementation decisions.

2.6 Case Study

In this section, we will demonstrate our exploitation of ION related vulnerabilities on a few latest and representative Android devices including flagship models from mainstream manufactures with newest Android system and kernel. It is worth noting that although the vulnerabilities usually manifest themselves differently on various devices due to customization, the underlying cause stems from the same design and implementation of ION as we outlined earlier.

2.6.1 DoS against Heaps of Fixed Size

Disable fingerprint authentication service on (multiple devices). On Nexus 6P, if one occupies enough free space of "qsecom" heap, which is of CMA type, the device's fingerprint authentication functionality will be effectively disabled. When a user tries to unlock the device with his/her finger, the system will show an error message "fingerprint hardware is unavailable". Similar attacks can also be performed on all Android devices using MSM platform, including Samsung Galaxy S7 and S6.

Disable audio service (multiple devices). On many MSM platform Android devices there exists an "audio" heap, with either CARVEOUT type or CMA type. If we exhaust this heap's free memory, the system will be unable to produce any sound, including ringtones. Affected devices, such as Nexus 6P and OPPO R7s, will be unable to notify users or play any music under this attack. The sound playback is the responsibility of a system service named "mediaserver", which heavily depends on the "audio" heap as its buffer provider on MSM-based Android devices.

System crash on Huawei Mate 8. Huawei Mate 8 is shipped with a CARVEOUT heap named "carveout-heap". We can request memory buffers from it and when we claim and keep a big enough buffer, the device will crash directly. This CARVEOUT heap has only a fixed size, and is used by the critical system framebuffer service that is responsible for the screen display and refresh. When we occupy too much resource in the heap, the critical system service will fail to work and eventually cause the whole system to crash.

2.6.2 DoS against Heaps of Unlimited Size

System level DoS (multiple devices). On virtually all devices, there is a SYSTEM heap usually named "system". If we allocate a large enough memory buffer (usually around or more than 1GB) from there, the system will freeze and many running background services will be killed at the same time, including music playback service and push notification service.

As mentioned previously in §2.3.1, SYSTEM heaps will request new pages from the basic buddy allocator when needed, so the available memory for them is equal to that of the whole system. Surprisingly, there is one important difference between allocating memory from the "system" heap and simply using functions like `malloc()`: the memory from the latter will be considered "owned" by the calling process, whereas the buffers from the former is actually allocated and "owned" by the ION driver; what the requesting process gets is only a handle to the buffer. Since Android is equipped with a low memory killer [2] which is responsible for releasing memory by killing processes when it detects that the system is currently low on memory, when a process allocates too much memory via interfaces like `malloc()`, it will gain a high priority in the "killing list" since the killer thinks that it owns too much, the result is that it will be killed soon and the system will thus recover quickly. However, when allocating a large amount of memory from the "system" heap, the killer will not consider our process as a main memory holder; instead, it will try to kill other innocent processes such as the push notification service.

2.6.3 DoS via Protected Memory Access

System crash (multiple MSM platform devices). There is a CMA heap named "pil" on Nexus 6P and many other Android devices that use MSM platforms, from which a program can request memory buffers. By trying to access the allocated buffer, the whole Android operating system will crash and the device will reboot immediately. The name "pil" is short for "peripheral loader"; it is used to load peripheral devices' firmware images when kernel boots. Since it is important to guarantee firmware images' integrity, the memory region of this heap will be protected by TrustZone, which is an SoC security extension providing Trusted Execution Environment (TEE) for sensitive operations with physically isolated memory and CPU mode [6]. Thus, any access to this protected memory region from normal world will cause the system to raise a protection exception, which usually leads to a system reboot.

The problem here is that a user can allocate buffers from "pil" heap even though it is never supposed to be exposed to user space. Unfortunately, as we highlight, the unified interface of ION grants a program access to virtually all heaps. In addition, the buffer sharing capability allows a program to further access the allocated buffer. Either reset the protection before the user can access the allocated buffers or simply deny users' memory requests for "pil" heap can solve the problem; however, neither is done on the devices.

2.6.4 Information Leakage

According to our analysis, there exist three different types of information leakage vulnerabilities, classified based on ION heap types. They are CARVEOUT, CMA and SYSTEM_CONTIG respectively, all of which do not zero the buffers before returning them to user space. This series of vulnerabilities can cause sensitive information leakage from both system and user applications, enabling an attacker to easily breach user privacy such as getting access to email content, bank accounts, and passwords.

Camera data leakage on Samsung Galaxy S7. There exists a CARVEOUT heap named "camera" in Samsung's newest flagship phone model S7. We can obtain image data captured by the phone's camera without any permission with the following steps:

(1) Open the system "camera" application, or third-party camera applications which also need to use the system camera hardware. Then capture some images with the

application, note that it is unnecessary to actually take a picture — simply seeing the preview on the screen is enough.

(2) Close the camera application and then dump the whole "camera" heap by having any app allocating buffers from it and read their content, which contains the image data captured by the system camera, including the previews. We confirm this by byte-to-byte comparison between the picture (taken by the camera) and the memory content of the ION buffers.

As shown in Table 2.1, a CARVEOUT heap will manage a fixed physically contiguous memory region which is reserved by kernel at boot time for special purposes (*e.g.,* to satisfy the requirement of certain hardware devices). On S7, "camera" is such a heap that serves as a data buffer used by the system camera service. The camera device must have the requirement of physically continuous memory in order to perform DMA (and possibly other) operations. When users are running a camera app, the camera device should be populating the image data into the buffer allocated from the "camera" heap, which will then be released when users exit the app. Thus, an attacker can now re-allocate buffers from the "camera" heap to obtain the dirty buffer.

In our research, this kind of problems for CARVEOUT heaps widely exist on multiple Android devices, enabling attackers to steal sensitive data from various system services. For example, the audio data can be leaked via "audio" heap on Nexus 6P and many other MSM-based Android devices in the same way.

Live memory dump of running apps (multiple devices). "Qsecom" heap on Nexus 6P is of CMA type, which is used mainly by TrustZone related services on MSM platforms.

Different from CARVEOUT heaps that are used exclusively by certain system services, with the CMA heap we can obtain various kinds of sensitive information, including but not limited to Gmail contents, Chase bank transactions and wi-fi passwords, from "live memory" of running apps. This is achieved by the following three steps:

(1) Drain free system memory by allocating as much memory as we can from either normal user space interfaces like malloc() or ION SYSTEM heaps as shown in §2.1. Note that the buffer allocation should not cause observable slowdown to the system to avoid alerting the users.

(2) Run any victim app (*e.g.,* Gmail) normally which will naturally produce sensitive data in memory.

(3) Dump the content of "qsecom" heap, which will contain sensitive information as mentioned above.

The root cause is as follows: while both CARVEOUT heaps and CMA heaps manage some pre-reserved contiguous memory regions dedicated for certain system services, a major difference is that CARVEOUT memory regions are set aside at system boot time and invisible to kernel memory manager after the system boots; thus no other processes can reuse these regions — even when they are free — using normal memory allocation interfaces (*e.g.,* `malloc()`) other than ION. Effectively, the CARVEOUT memory is stolen from the system, which guarantees the availability of memory buffers to the corresponding system service and hardware device (*e.g.,* camera) In contrast, CMA heaps expose their reserved memory regions to kernel memory manager (*e.g.,* Linux buddy allocator) and thus allow other processes to utilize these regions through standard interfaces such as `malloc()`, when there is no sufficient memory that can be found elsewhere (attack step 1 ensures this condition). However, to allow system services to function, the memory allocated from CMA heap can be reclaimed on demand as soon as the system service asks for them (which is how attack step 3 can successfully dump live memory of other running apps). Prior to the memory being reclaimed, the data generated on the CMA heap will be copied elsewhere and page tables will be updated to reflect the change. As we can see, the design of CMA heap allows a better utilization of memory resources at the cost of a potentially longer latency when memory is allocated from CMA heap [101].

The CMA attack is considered extremely dangerous since it effectively allows a malicious app to dump the live memory of any other apps. Equipped with the capability of knowing which apps run in the foreground (through attacks such as [54]), it can effectively be tailored to reliably extract any app-specific information. One may even be able to reconstruct the app GUIs by taking multiple snapshots of the memory dumps; This very attack was recently achieved by forensically dumping the entire physical memory [110]. Finally, even though we have not attempted, there is no reason to believe that it is impossible to extract crypto keys used by apps and the system.

System-wide information leakage on Mate 8 The "system-contig" heap on Huawei Mate 8 is assigned the type "SYSTEM_CONTIG". With this heap we can get various sensitive information similar to what we can get with CMA heap; they include Gmail content, Chrome browsing history and html data of previously loaded web pages. Besides data leaked from user applications, it is also possible to learn information from the kernel with such a heap type. The attack process is sketched as below:

(1) Open arbitrary apps and operate normally, then exit (which means that memory will be released by the apps).

(2) Allocate as much memory as possible from “system-contig” heap and record their content, which will include significant sensitive information.

Since SYSTEM_CONTIG heap draws pages directly from the basic system memory manager (*i.e.,* buddy allocator), whoever returns pages to the system without zeroing them can leak data to the attacker. This includes any application or kernel data such as passwords, credit cards, and secret keys. This vulnerability is very much similar to the one in CMA heap except that the opportunity arises only when the memory is freed by other applications.

2.7 Defense Discussion

The fundamental problems with ION stem from its two design goals, *unified interface* and *buffer sharing across layers*, as we highlight throughout this chapter. Since they are not really simple implementation errors, they require a more systematic investigation. One may consider the vulnerability of unzeroed buffers simply a glaring error; we argue that it actually is a much more complex problem than what it appears. As shown in §2.3.2.1 and §2.5, the unzeroed buffers are introduced due to several complex reasons:

(1) Kernel memory allocation functions have complex behaviors. Many of them never need to be exposed to user space prior to ION’s cross-layer buffer sharing capability. It is hard to make the correct assumption on whether a particular function will zero the returned buffer or not.

(2) Customization of ION can lead to a drastically different implementation from the common branch in AOSP. In our analysis of Nexus 6P, we find that its ION implementation follows the one customized by Qualcomm (due to the fact that Nexus 6P uses the Qualcomm SoC). Interestingly, we note that the Qualcomm kernel source was forked prior to AOSP common branch fixing the vulnerability of unzeroed buffers in CMA heap. Unfortunately, once the Qualcomm source tree is forked, it no longer merges the patches applied to AOSP common branch. This is demonstrated by the fact that even the Android 7.0 preview on Nexus 6P still has unzeroed buffer vulnerabilities.

Therefore, we believe the static taint analysis tool can be effective in assisting developers with the insight into the buffer zeroing behaviors and help them navigate the complex kernel functions.

However, even when the buffer zeroing problem is resolved. The DoS vulnerabilities introduced by ION's *unified interface* design still remain. Fundamentally, the unified interface hurts security as it supports only coarse-grained access control (through file permission of */dev/ion*) — a user has access to either all heaps or none. A fine-grained access control is necessary to solve the problem. For instance, a third-party app should not be able to access the heap used by fingerprint service; we have not seen a case where such heaps are accessed by any other process. Due to the fact that ION is a complex system and the entire software stack is involved (from applications, system services, and kernel drivers), it is important that the changes minimize side effects such as backward compatibility and performance hit. In addition, we should avoid adding new security mechanisms to the kernel and instead try to piggyback on existing mechanisms offered by Android and Linux if possible.

One straightforward solution can be adding the fine-grained access control to the kernel to govern how much memory each user can allocate (based on uid/gid). This solution maintains backward-compatibility to existing applications and system services (and can be implemented efficiently). However, the downside is that the kernel needs to maintain an access control list for all (uid, ion heap) pairs that does not fit in any existing Android or Linux security mechanism. Even SeAndroid/SeLinux cannot express such a security policy, as it will need to be able to interpret the argument `struction_allocation_data*` of `ioctl()` to extract the heap information. Furthermore, the additional access control list needs to be changed every time when customization occurs as heap types and instances may change.

Alternatively, one can place the access control enforcement at the user space. This requires revoking direct access to */dev/ion* from third-party apps and allowing access to only system services. The idea is that apps will need to go through system services to allocate memory from ION heaps, in which case the system services can enforce the access control policy. For instance, if the policy says that a regular app can allocate memory from SYSTEM heap for graphics processing of up to 20MB, it will have to send the request to "mediaserver" (or whichever process that is responsible for managing graphics buffers). The "meidaserver" can then check if the requested memory is indeed in SYSTEM heap (not any other heap). Further, it will keep track of how much memory has been used by the user. This solution requires grouping all system service uids (*e.g.,* media) into an "ion" group so that the file permission of */dev/ion* can be changed to 660 (`rw-rw----`) where the user owner is system and group owner is ion. Now an app needs to go through system services for memory allocation in ION heaps. Since most apps only need to request memory for graphics buffers, the changes should involve only the app-side libraries that are responsible for allocating graphics buffers (that previously interacts with */dev/ion* directly) and a single system service (*e.g.,* "mediaserver"). Other heaps are automatically inaccessible to third-party apps. This does introduce the overhead of an additional IPC round trip for each ION memory allocation. Note that even though the memory allocation goes through system services, the returned memory pages still need to be zeroed. With this extra layer of indirection, the zero operation can in fact be performed by the system service itself before sending the file descriptor over to the app that requested the memory.

Besides the challenge in maintaining backward-compatibility, another downside of the alternative design is that once a process capable of accessing */dev/ion* is compromised, it can still launch the DoS attacks against other services. That is because of the access control not fine-grained enough to differentiate different system services. To truly achieve fine-grained access control, each system service needs to run with a different uid which the kernel can use to enforce the access control properly (as is done in the first solution).

In summary, we show two potential solutions that have different tradeoffs in the following aspects: backward-compatibility, performance, and avoid introducing new security mechanisms in the kernel.

2.8 Related Work

Android customization and related security issues. Android customization is known to introduce new security vulnerabilities across layers. At the application layer, pre-loaded apps have been shown to require more permissions than needed [125]. At the framework layer, customized system services have been shown to have missing permission checks [111]. At the system layer, devices files are shown to have weak permissions that allow third-party apps to directly manipulate device drivers and perform privileged operations [134]. In addition, devices drivers also introduce vulnerabilities that can directly cause root exploits [132]. A recent study has shown that by analyzing configuration differences across customized Android ROMs, many security flaws can be reveled in all these layers [42]. Our study is a systematic analysis of an overlooked system component, Android ION, that is customizable by SoC and smartphone vendors.

Android DoS vulnerabilities. DoS attacks (*e.g.,* soft reboot) against the Android system services have been demonstrated using different techniques, *e.g.,* by issuing targeted and repeated requests to the system services [80], or forking an unlimited number of processes exploiting a weak local socket permission of the Zygote process [43]. In addition, a number of other vulnerabilities such as Null pointer and integer overflow have been reported recently [19, 3]. All of the DoS attacks can cause only the entire Android framework or system to reboot. Our DoS attacks exploit a new class of vulnerabilities that exist due to the lack of access control and memory usage limit in various ION heaps. Any services or apps that require memory from ION can be targeted. Due to the fact that some heaps are mostly used by one or two services, the DoS impact can be controlled to affect only those services; this has not been reported before.

Unzeroed/Dirty memory. Dirty memory can leak critical data to a malicious application, and there are different underlying causes. For instance, recently it has been shown that newly allocated GPU memory pages are not zeroed, and may contain data rendered by other applications [92]. In Linux, memory obtained by `malloc()` will automatically be zeroed by the underlying OS if a physical page has been previously used by a different process [64]. However, kernel memory allocation functions like `kmalloc()` do not get zeroed as they are intended for kernel-space use only; this has obvious performance benefits. Unfortunately, the introduction of ION and its user-space and kernel-space buffer sharing capability effectively breaks the assumption.

Static analysis tools on Android. Static taint analysis is one of the most popular techniques used to analyze and vet Android apps [98, 97, 45, 71, 72, 124]. For instance, Chex [98] can statically analyze the byte code of Android apps for component hijacking vulnerabilities. AAPL [97] compares the the produced information flows for apps in similar categories (*e.g.,* news) to identify suspicious apps with excessive information flows compared to others. Besides taint analysis, many other static analysis tools are built to discover vulnerabilities. Woodpecker [74] analyzes apps to look for capability leaks (*e.g.,* through Intent) that allow confused-deputy attacks. At the system layer, Kratos [111] analyzes Android framework and look for inconsistent security policy enforcements. Even though static taint analysis has been used widely to analyze apps, it is rarely used to analyze the

Android/Linux kernel. Our work applies static taint analysis in a novel setting to identify unzeroed memory pages allocated and returned to user space through ION.

2.9 Conclusion

In this chapter, we report multiple vulnerabilities of the ION memory management system that can lead to either DoS or sensitive information leakage on virtually all Android devices to date. We build a novel static taint analysis tool to uncover the unzeroed ION heap vulnerabilities systematically. To demonstrate the seriousness of the vulnerabilities, we build exploits against several latest Android devices running latest Android operating systems, including Nexus 6P, Samsung Galaxy S7, and Huawei Mate 8 that run Android 6.0 and even 7.0 preview. In addition, we analyze and digest the root causes of the vulnerabilities in depth. Finally, we outline the defense strategies that have different tradeoffs which can shed light on future design of such a large and complex memory management system.

Chapter 3

Improve Kernel Testing: Statically Detect High-Order Taint Style Vulnerabilities

3.1 Introduction

A large amount of vulnerabilities can be classified into the category of taint-style vulnerability [128], where an attacker-controlled input flows into some sensitive or dangerous operations (*e.g.,* the input used as an array index without sanity check can cause an out-of-bound access) in the program. Thus taint analysis has become an important technique for vulnerability discovery.

Though many taint style vulnerabilities are in the straightforward "one-shot" form (*i.e.,* the control flow from the taint source to sink is confined within a single entry function invocation), in more stateful software (*e.g.,* Linux kernel), there exists more complicated cases where the taint flow crosses multiple entry function invocations, *e.g.,* entry function `A` copies its user-provided argument to a global variable `G` in one path and then another entry function (`A` or not) invocation performs a dangerous operation involving `G`, we define the latter as **high-order taint style vulnerabilities**, where the "order" equals to the times of required entry function invocations to trigger the bug.

Due to the complex taint flow, high-order taint vulnerabilities are more stealthy and difficult to be identified by both human experts and automatic tools. There are many existing works on applying and improving taint analysis for vulnerability detection, however, few of them consider the high-order taint vulnerability detection problem, or only try to address the problem in a specific setting with limited order support [62].

In this chapter, we develop a novel static taint analysis capable of detecting high-order (our tool supports arbitrary order) taint vulnerabilities in the Linux kernel effectively and efficiently, to achieve this goal, we have two main challenges:

(1) Our taint analysis needs to be highly precise and accurate. Since the high-order taint flows are usually "concatenated" from multiple flows local to individual entry functions, analysis inaccuracies will accumulate and eventually cause an unacceptable amount of false alarms. Thus, some common compromises (*e.g.,* flow- and field- insensitivity) in static analysis may not apply. To address this challenge, we develop an inter-procedure flow-, context-, field-, index-, and partially path- sensitive static multi-tag taint analysis based on LLVM IR [36]. To better support the kernel analysis, we also make considerable

efforts handling kernel code patterns (*e.g.,* pointer arithmetic). To our best knowledge, we are unaware of existing static taint analysis for kernel sharing the same level of accuracy.

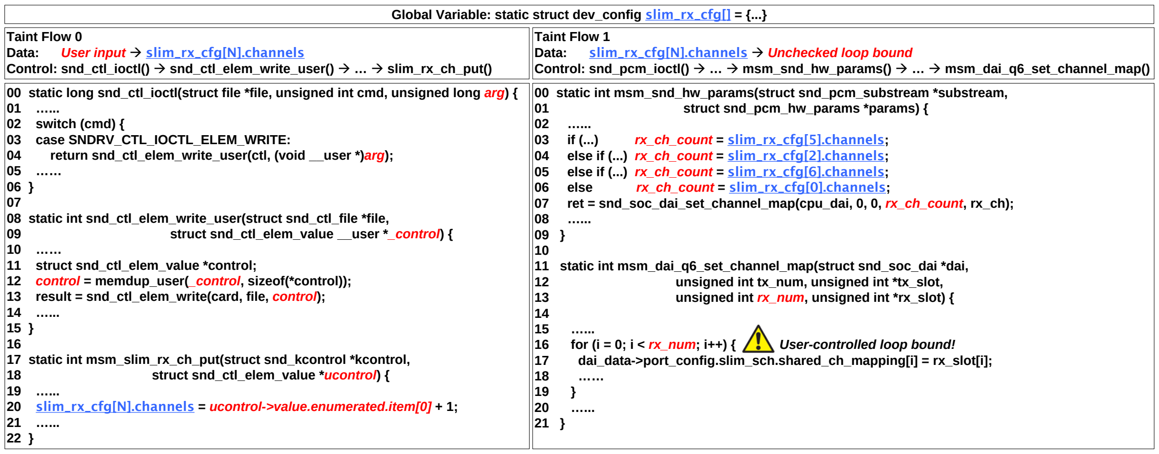
(2) We need an efficient method to analyze all possible cross-entry-function taint flows. Intuitively, since these entry functions can be invoked in any order, a sound analysis needs to enumerate every possible order and repeatedly analyzes a same entry function in different calling sequences (*e.g.,* a global pointer used in entry function `A` may be set to different values in entry function `B` and `C`), which can be very inefficient if not infeasible. To tackle this problem, our core idea is to first analyze each entry function independently and create an abstract summary regarding its taint behaviors involving both local and global variables, then in the vulnerability detection phase, we try to re-construct all possible attacker-controlled high-order taint flows by querying the individual summaries - a much faster process than the aforementioned naive method.

The evaluation shows that our tool can both confirm known high-order taint vulnerabilities and discover new ones in the kernel modules, with a low reviewer conceived false positive ratio. We summarize our contributions as below:

(1) To our best knowledge, we are the first to try to automatically detect high-order taint style vulnerabilities in the kernel. Our method can also be easily generalized to other stateful software.

(2) We implement a prototype tool SUTURE, which is able to do high-accuracy point-to and taint analyses for the kernel and construct high-order taint flows. We are also planning to open source our tool.

(3) We successfully detect new high-order taint vulnerabilities in the kernel.



Global Variable: static struct dev_config slim_rx_cfg[] = {...}

Taint Flow 0
Data: User input → slim_rx_cfg[N].channels
Control: snd_ctl_ioctl() → snd_ctl_elem_write_user() → ... → slim_rx_ch_put()

```
static long snd_ctl_ioctl(struct file *file, unsigned int cmd, unsigned long arg) {
  ......
  switch (cmd) {
  case SNDRV_CTL_IOCTL_ELEM_WRITE:
     return snd_ctl_elem_write_user(ctl, (void __user *)arg);
  ......
}

static int snd_ctl_elem_write_user(struct snd_ctl_file *file,
                                   struct snd_ctl_elem_value __user *_control) {
  ......
  struct snd_ctl_elem_value *control;
  control = memdup_user(_control, sizeof(*control));
  result = snd_ctl_elem_write(card, file, control);
  ......
}

static int msm_slim_rx_ch_put(struct snd_kcontrol *kcontrol,
                              struct snd_ctl_elem_value *ucontrol) {
  ......
  slim_rx_cfg[N].channels = ucontrol->value.enumerated.item[0] + 1;
  ......
}
```

Taint Flow 1
Data: slim_rx_cfg[N].channels → Unchecked loop bound
Control: snd_pcm_ioctl() → ... → msm_snd_hw_params() → ... → msm_dai_q6_set_channel_map()

```
static int msm_snd_hw_params(struct snd_pcm_substream *substream,
                             struct snd_pcm_hw_params *params) {
   ......
   if (...)       rx_ch_count = slim_rx_cfg[5].channels;
   else if (...)  rx_ch_count = slim_rx_cfg[2].channels;
   else if (...)  rx_ch_count = slim_rx_cfg[6].channels;
   else           rx_ch_count = slim_rx_cfg[0].channels;
   ret = snd_soc_dai_set_channel_map(cpu_dai, 0, 0, rx_ch_count, rx_ch);
   ......
 }

 static int msm_dai_q6_set_channel_map(struct snd_soc_dai *dai,
                          unsigned int tx_num, unsigned int *tx_slot,
                          unsigned int rx_num, unsigned int *rx_slot) {

    ......
    for (i = 0; i < rx_num; i++) { ⚠ User-controlled loop bound!
      dai_data->port_config.slim_sch.shared_ch_mapping[i] = rx_slot[i];
      ......
    }
    ......
 }
```

Figure 3.1: Vulnerable Code Excerpt of CVE-2017-0608

3.2 Overview

In this section we describe the architecture and overall workflow of SUTURE, with a motivating example.

3.2.1 A Motivating Example

We use CVE-2017-0608 [12], a high severity vulnerability in qualcomm sound driver, as a motivating example of high-order taint style vulnerability. We show the simplified vulnerable code excerpt in Figure 3.1, to exploit the vulnerability, the attacker first needs to invoke the entry function `snd_ctl_ioctl` to set a global variable `slim_rx_cfg[N].channels` with the user provided input (*i.e.*, the `arg` argument of `snd_ctl_ioctl`) - we call this process taint flow 0 as shown in the left side of Fig 3.1, then the invocation of the second entry function `snd_pcm_ioctl` will propagate the value of the global variable `slim_rx_cfg[N].channels` eventually to an unchecked loop bound `rx_num` in taint flow 1 on the right side of Fig 3.1. As we mentioned earlier in §3.1, this vulnerability involves 2 entry function invocations (and 2 taint flows, respectively), so it is a 2-order taint style vulnerability.

Previous taint analysis methods are unable to detect such high-order taint vulnerabilities because they only focus on the individual local taint flows but fail to stitch them together, as shown in Fig 3.1, if we only look at taint flow 1, though the unchecked `rx_num` is used as a dangerous loop bound, we have no evidence that its value can be controlled by the user, so no warnings will be issued, only after we also take taint flow 0 into the picture, can we confirm that the dangerous loop bound can be (indirectly) manipulated by the user, making it a severe vulnerability.

3.2.2 System Architecture and Workflow

We provide an overview of our solution to discover high-order taint style vulnerabilities (*e.g.,* Fig 3.1) in this section. The architecture of SUTURE is shown in Fig 3.2. Since our static analysis is based on LLVM, we require the input program to be compiled into LLVM bitcode, together with a config file specifying the entry function information (*e.g.,* function name, user-controlled argument) of the input program. SUTURE will then perform static taint analysis and generate summary for each entry function, note that in this phase both user input and all global variables are treated as taint source to ensure the recovery of every possible high-order taint flow later, regardless of the entry function analysis order (*e.g.,* in Fig 3.1 if `snd_pcm_ioctl` is analyzed first without setting the global variable as taint source, taint flow 1 will be invisible). After the taint summary has been generated, each entry function will be traversed again with various vulnerability detectors

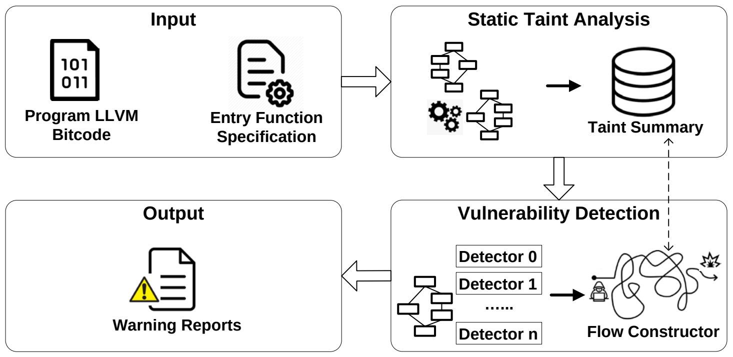


Figure 3.2: System Architecture of SUTURE

which can detect different types of bug-inducing program statements (*e.g.,* arithmetic calculation may cause integer overflow), SUTURE will then decide whether there are any user-controlled variables involved in these statements, by inspecting whether there exists a taint flow (including the high-order ones) from the user input to the variable (*i.e.,* "Flow Constructor" in Fig 3.2), if any, a warning will be issued.

For the specific example in Fig 3.1, after performing the taint analysis on both entry functions, SUTURE first summarizes that the global variable `slim_rx_cfg[N].channels` is tainted by a user input in the entry function `snd_ctl_ioctl`, while the loop boundary `rx_num` is tainted by `slim_rx_cfg[N].channels` in the second entry function `snd_pcm_ioctl`. Later in the vulnerability detection phase, the sensitive loop condition is recognized by one detector, so the flow constructor then tries to construct a user-input originated taint flow for the involved variable `rx_num`, by querying the records in the taint summary, it can successfully and efficiently construct the 2-order taint flow to `rx_num` as mentioned in §3.2.1 and fire a warning.

3.3 System Design

In this section we describe the design of SUTURE.

3.3.1 Definitions

We first formally define some concepts related to the design of SUTURE.

Def 0 An entry function ε of a program module (*e.g.,* a kernel driver module) serves as a module interface, thus it does not have any callers within the same module and intends to be invoked directly by the user or other modules (*e.g.,* top-level `ioctl`s of a driver).

Def 1 The taint source $\mathbb{S}$ includes both user-provided arguments of entry functions and all global variables, more formally:

$$\mathbb{S} = \mathbb{U} \cup \mathbb{G}$$

$$\mathbb{U} = \{v | \exists \varepsilon, v \text{ is a user argument of } \varepsilon\}$$

$$\mathbb{G} = \{v | v \text{ is a global variable}\}$$

Def 2 A calling context $\triangle$ is defined as a sequence of instructions:

$$\triangle = [i_0, i_1, ..., i_{2n}]$$

An instruction with an even subscript denotes the entry instruction of a caller function, while the odd denotes a call site instruction within the caller (*e.g.,* i_2 is the entry of the function that is called at i_1), the sequence always ends with the entry instruction of current executing function, so its length is always odd. This definition enables us to differentiate multiple callees at a same call site (*e.g.,* an indirect call with multiple potential targets).

Def 3 We define an "instruction location" (*InstLoc* for shorter in the remaining chapter) as an instruction plus the calling context it is executed in, we use I to denote an $InstLoc$ to differentiate it with a static instruction i:

$$I = (i, \triangle)$$

Def 4 A taint flow τ is basically an $InstLoc$ sequence:

$$\tau = [I_0, I_1, ..., I_n]$$

The first *InstLoc* I_0 initiates the taint propagation from one taint source variable $v \in \mathbb{S}$, while the remaining $InstLoc$s pass the taint on.

Def 5 We define the connect operator $\circ$ for taint flows as following:

$$\tau_0 = [I_{00}, I_{01}, ..., I_{0n}], \tau_1 = [I_{10}, I_{11}, ..., I_{1n}]$$

$$\tau_0 \circ \tau_1 = \begin{cases} [I_{00}, ..., I_{0n}, I_{10}, ..., I_{1n}], & \text{if sink}(I_{0n})\text{==src}(I_{10}) \\ \varnothing, & \text{else} \end{cases}$$

Basically two taint flows can be sequentially connected *iff* the last $InstLoc$ of one taint flow propagates the taint to a variable that is used as the taint initiator at the beginning of the other taint flow.

Def 6 We now define the "order" of a taint flow with the $order()$ function, before that, we need to first define the $reach()$ function to test the reachability between two $InstLoc$s:

$$reach(I_0, I_1) = \begin{cases} True, \text{if } \exists\varepsilon,\ I_0 \text{ can reach } I_1 \text{ on } ICFG(\varepsilon) \\ False, \text{else} \end{cases}$$

$ICFG(\varepsilon)$ means the inter-procedure control flow graph of the entry function ε, I_0 can reach I_1 on $ICFG(\varepsilon)$ implies that there is one execution of ε that can reach the $InstLoc$ I_0 and after that, I_1. With the $reach()$ definition:

$$order(\tau) = |\{k | I_k \in \tau, I_{k+1} \in \tau, \neg reach(I_k, I_{k+1})\}|$$

Intuitively, $order(\tau)$ is the number of "breakpoints" in τ, where to continue following the taint flow we have to make another entry function invocation. We hereby call a taint flow τ **high-order taint flow** if $order(\tau) > 1$.

It is worth noting that since $reach()$ is not transitive (*e.g.,* $reach(I_0, I_1) \wedge reach(I_1, I_2) \nrightarrow reach(I_0, I_2)$, because the path between I_0 and I_1 may pose conflicting constraints to that between I_1 and I_2), our definition of $order()$ can underestimate the real taint flow order (*i.e.,* there can be more "breakpoints" in a taint flow than counted by $order()$), however, to calculate the real order may be too difficult and expensive due to the constraint solving, while our definition makes the order calculation practical and simpler. Besides, we find that the underestimation rarely happens in practice.

Def 7 We define the local taint flow set of an entry function ε as LT_ε, it is the set of all taint flows that can be reproduced within one invocation of ε, naturally, we have $\forall \tau \in LT_\varepsilon, order(\tau) == 1$.

3.3.2 Input

SUTURE needs two kinds of input: the to-be-analyzed program compiled to LLVM bitcode and its entry function specification, the latter manifests all the entry function names to analyze and the user-controlled arguments of each function (*i.e.,* $\mathbb{U}$), note that $\mathbb{U}$ cannot be $\emptyset$, otherwise no meaningful taint vulnerabilities can be detected. This input design gives us some flexibilities when analyzing large programs, since we can only compile a part of the program that contains just the required entry functions (*e.g.,* instead of compiling a whole kernel we can only do a single driver module), or only specify a set of mutual-related (*e.g.,* access same global variables) entry functions in the program to analyze. Note that the entry function specification can either be generated from automatic pattern matching based on some domain knowledge for specific programs (*e.g.,* as done for kernel drivers in [99]), or from manual inspection.

3.3.3 Static Taint Analysis

The goal of static taint analysis is to construct a taint summary (detailed in §3.3.3.8) for each entry function ε, to achieve this goal, we independently analyze each ε and record its taint facts. Our static taint analysis follows the basic design and reuses the main data structures of Dr.Checker [99], which makes a soundy inter-procedure traversal of each ε in the top-down style and for each visited LLVM IR, invokes the alias and taint analysis clients (the latter relies on the former) to process it, updating the point-to or taint information associated with variables involved in the IR. The rules for point-to record update and taint propagation are standard as manifested in [99]. Dr.Checker's static analysis is context- (partially, as detailed in §3.3.3.2), flow- (partially, as detailed in §3.3.3.2), and field- (limited, as detailed in §3.3.3.5) sensitive, for brevity, we will omit its technical details here. Since Dr.Checker cannot satisfy our requirements in both functionality and accuracy, in this section, we discuss our enhancements over it.

3.3.3.1 Multi-Tag Taint Analysis

As mentioned in §3.2, to construct high-order taint flows we must be able to differentiate multiple taint sources (see **Def 1,5** in §3.3.1), *e.g.,* in Fig 3.1 we must know exactly that taint flow 1 originates from the specific global variable `slim_rx_cfg[N].channels` to connect it to taint flow 0. However, Dr.Checker only maintains the binary "tainted or not" state without differentiating the taint sources, to address this problem we turn the original taint analysis into a multi-tag one - for $\forall v \in \mathbb{S}$ we will have a unique taint tag associated, which will also be propagated to all tainted variables, enabling us to easily query the taint sources for each τ.

3.3.3.2 Memory SSA based Analysis

The accuracy and correctness of Dr.Checker's static analysis is significantly limited when dealing with the address-taken memory objects in LLVM IR, this is mainly because when updating the point-to record and taint state for a certain variable, Dr.Checker ignores the location information (*e.g.,* at which instruction and under which calling context is the point-to relationship established) - while this works for the top-level variables since inherently LLVM IR put them in the SSA (Static Single Assignment) [109] form (in other words, the identifier of a top-level variable already indicates its sole definition location), it does not for the address-taken memory objects. Consequently, context- and flow- sensitivity are indeed not enforced for the memory objects - both false positives and false negatives can occur regarding the point-to and taint analysis results. To solve this problem, we bring the memory SSA form [56] to the original analysis. Basically, we append an $InstLoc$ to each point-to/taint update to represent its location, and correctly handle the point-to/taint merge and overwrite for memory objects according to the location information, in a standard way (*e.g.,* if the location of a strong point-to update post-dominates a previous one on a same memory object field, the old point-to should be masked out from the new location).

3.3.3.3 Index Sensitivity

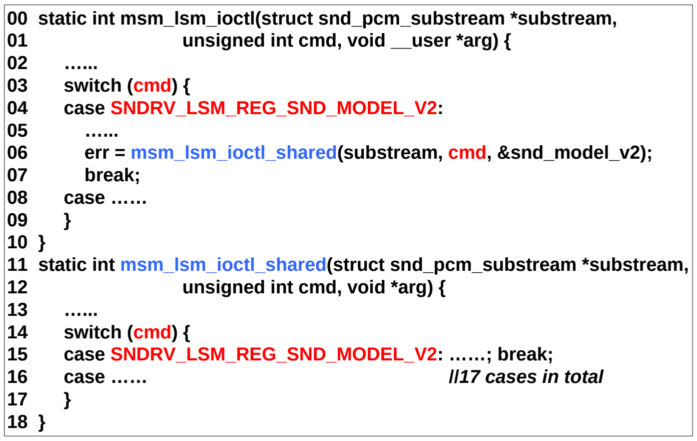
We add the index-sensitivity for array access to Dr.Checker to improve the analysis accuracy. In principle, our design of index-sensitivity follows two rules for array read/write respectively:

(1) If an array element is read with a constant index (*e.g.,* v = a[2]), we return the point-to/taint record related to exactly that index; If the index is a variable (*e.g.,* v = a[i]), we conservatively merge the records of all array elements and return them.

(2) If an array element is written with a constant index (*e.g.,* a[2] = v), we perform a strong update (*i.e.,* the new records can overwrite the old ones) for exactly that index; If the index is a variable (*e.g.,* a[i] = v), we conservatively update every array element, and the update is weak (*i.e.,* new records co-exist with old ones).

3.3.3.4 Partially Path Sensitivity

It is well known that static analysis (especially the path-insensitive ones) can follow infeasible paths since the unawareness of conflicting path constraints, this can cause both inaccuracy (*e.g.,* impossible taint propagation in reality) and inefficiency (*e.g.,* some code traversal should have been avoided). The straightforward alleviation is to make the analysis path-sensitive, however, a fully path-sensitive analysis (*e.g.,* symbolic execution) can be



```
00 static int msm_lsm_ioctl(struct snd_pcm_substream *substream,
01                    unsigned int cmd, void __user *arg) {
02    ……
03    switch (cmd) {
04    case SNDRV_LSM_REG_SND_MODEL_V2:
05       ……
06       err = msm_lsm_ioctl_shared(substream, cmd, &snd_model_v2);
07       break;
08    case ……
09    }
10 }
11 static int msm_lsm_ioctl_shared(struct snd_pcm_substream *substream,
12                    unsigned int cmd, void *arg) {
13    ……
14    switch (cmd) {
15    case SNDRV_LSM_REG_SND_MODEL_V2: ……; break;
16    case ……                                    //17 cases in total
17    }
18 }
```

Figure 3.3: An example of partially path-sensitive analysis

very expensive (*e.g.,* complex constraint solving and path explosion), so to take advantages of path-sensitive analysis while avoiding the high expense, we propose the partially path-sensitive analysis.

The basic idea is that we collect and solve only a limited set of simple path constraints in the form of $v\ op\ C$, where v is a variable, C is a constant (*e.g.,* a literal number), and $op \in \{==, >, <, \geq, \leq\}$. Since these simple constraints are widely spread in the programs but easy to solve, we can filter out a considerable amount of infeasible paths (and false positive analysis results related to them) without significantly impacting the analysis performance - in fact, our top-down style analysis actually greatly benefits from it regarding the performance because of the avoidance of infeasible code (*e.g.,* the analysis time of the sound driver module decreased from 54 hrs to 31 hrs). We show one example in Fig 3.3 to demonstrate our partially path-sensitive analysis, as we can see, `msm_lsm_ioctl` calls `msm_lsm_ioctl_shared` at line 6, under one specific switch-case with the `cmd` restricted to `SNDRV_LSM_REG_SND_MODEL_V2`, the same `cmd` is passed to the callee and used again as the switch conditional at line 14, since its value has already been restricted at the call site (line 6), there is actually only one valid switch-case in the callee (line 15) under this calling context. Our partially path-sensitive analysis can collect the equality constraint on `cmd` at line 6 and propagates it to the callee `msm_lsm_ioctl_shared`, where we can filter out the infeasible switch-cases (16 out of 17) due to the conflicting constraints, improving both accuracy and efficiency.

3.3.3.5 General Language Feature Support

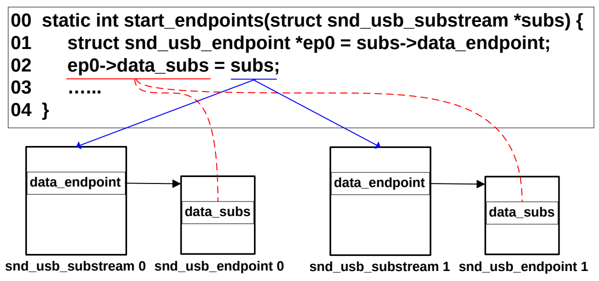
There are several general LLVM language features that are not supported or well handled by Dr.Checker, in this section we discuss two most important ones.

Nested Structure. Nested structure (*i.e.,* one structure is embedded as a field in a parent structure) is a widely used feature and failure to correctly handle it can significantly impact the analysis accuracy, we therefore add the support to nested structure in SUTURE. More specifically, Dr.Checker only differentiates the top-layer structure fields but not the sub-layer ones within an embedded structure (*i.e.,* the embedded structure is not expanded), making its field-sensitivity limited. To address this issue, we re-design the data structures to represent a memory object, supporting nested structures of arbitrary layers by recursively creating a new abstract memory object for each embedded field, we also re-write most IR processing logics in the point-to and taint analysis to take nested structures into account (*e.g.,* process all indices of the *GEP* instruction instead of only the first 2 by Dr.Checker).

Pointer Arithmetic. It is well known that pointer arithmetic in the C language family can often cause inaccuracies in static analysis, since it is difficult to keep track of the exact pointer location during the arithmetic calculation, *e.g.,* normally, LLVM IR accesses the *2nd* field of a structure by simply specifying the field number *2* in the *GEP* instruction with the structure base pointer, however, in some cases (*e.g.,* optimization) the field can be accessed by directly subtracting an offset (between the *2nd* and *5th* fields) from the pointer to the *5th* field. To handle such cases, SUTURE keeps a record of detailed layout information (*e.g.,* size and offset of each field in bytes) of each structure and faithfully calculates the new target field after pointer arithmetic according to the pointer type (*e.g.,* pointer conversion aware, like `(char*)p-1` and `(int32*)p-1` are different), offset to add/sub, and the structure layout. It is worth noting that our pointer arithmetic handling has a byte-level accuracy and we always try to align to the field boundary, though rare, this may cause some inaccuracies (*e.g.,* a pointer points to the middle of a field, or bit-level pointer arithmetic), we leave the handling of these cases as the future work.

3.3.3.6 Kernel Code Pattern Handling

Although high-order taint style vulnerabilities are not specific to the kernel, SUTURE by design can also be extended to other general software, our main goal in this chapter is to hunt high-order taint vulnerabilities in the kernel, so the same as Dr.Checker, we also need to take care of some special kernel code patterns. From this perspective, our main improvement over Dr.Checker is the indirect call resolution, which is an important topic for kernel code analysis since the indirect call is prevalent in kernel. Dr.Checker uses the type-based method to resolve indirect call targets, though it is a common and standard solution, it can cause many false positives in practice, to further improve the accuracy and efficiency, SUTURE employs a method similar to PeX [133], which takes advantages of



```
00  static int start_endpoints(struct snd_usb_substream *subs) {
01      struct snd_usb_endpoint *ep0 = subs->data_endpoint;
02      ep0->data_subs = subs;
03      ......
04  }
```

Figure 3.4: An example of multi-source multi-sink pairing

domain knowledge on the kernel coding paradigm and resolve the indirect call targets by matching the parent structure and field id of the function pointer.

3.3.3.7 Multi-Source Multi-Sink Pairing

One unique challenge for static analysis we identified during our study is the multi-source multi-sink pairing problem. Fig 3.4 shows an example of this problem, when starting analyzing the function `start_endpoints`, the argument `subs` is a pointer that points to two instances of `snd_usb_substream` according to the previous static analysis results. Consequently, the left side of the assignment at line 2 can be one of two memory locations (*i.e.,* `data_subs` field of `snd_usb_endpoint`, either instance 0 or 1), while the right side `subs` points to either instance 0 or 1 of `snd_usb_substream`. In this situation, the common way to perform the assignment (as used in many popular static analysis tools like Dr.Checker [99] and SVF [117]) is all to all (*e.g.,* `data_subs` field of `snd_usb_endpoint` 0 will point to both `snd_usb_substream` 0 and 1). However, it is obvious that in the real program execution, `data_subs` can only point back to its own parent `snd_usb_substream` structure (*e.g.,* 0 to 0 and 1 to 1). We call this multi-source multi-sink pairing problem, failure to pair the two sides (*e.g.,* all-to-all update) will create many false positives in the analysis results.

To solve this problem, our key observation is that in the aforementioned scenario, two sides of the assignment actually share the same source of multiplicity (*e.g.,* the left side `ep0->data_subs` at line 2 has two possible locations because `ep0` can point to two structure instances, which is again because `subs` is so at line 1, the same reason for the right side), thus, as long as the unique source "collapses" to one of many possibilities in the runtime, both sides of the assignment will also be restricted. Following this observation, for every LLVM IR that can serve as a "source of multiplicity" (*e.g.,* a *phi* instruction can aggregate multiple point-to/taint records from different paths to its receiver variable), SUTURE will uniquely label every individual outcome record, and propagate the label to its derived records (*e.g.,* in Fig 3.4 the pointer `ep0` is derived from `subs`, the latter's two point-to records for `snd_usb_substream` 0 and 1 have their labels respectively inherited by `ep0`'s two records for `snd_usb_endpoint` 0 and 1), by matching these labels, when the multi-to-multi assignment happens we can correctly pair the source and the sink if they share the same source of multiplicity.

3.3.3.8 Taint Summary Construction

So far we have described our multiple enhancements on static taint analysis over Dr.Chekcer, all these efforts are for a same goal: to construct an independent taint summary for each entry function ε as accurate as possible. The taint summaries allow SUTURE to efficiently recover high-order taint flows in the later vulnerability detection phase, regardless of the analysis order of entry functions. This is also a major difference between SUTURE and Dr.Checker - the latter does not have the per-entry taint summary and the vulnerability detection is sensitive to the analysis order (as discussed in §3.2.1).

The taint summary for entry function ε is basically its local taint flow set LT_{ε} (**Def 7** in §3.3.1), in other words, the summary records all local taint flows originating from $\mathbb{S}$ and sinking to every accessed variable (local or global) within $ICFG(\varepsilon)$. SUTURE organizes these taint flows by the sink variable - each sink variable is mapped to a set of τ sinking to it, while the source variable can be obtained via the taint tag (See §3.3.3.1) associated with each τ. This enables a quick query of τ by sink, as well as the connect operation (**Def 5** in §3.3.1) for constructing high-order taint flows.

3.3.4 Vulnerability Detection

After generating the taint summary for each ε, SUTURE can then proceed to detect the high-order taint style vulnerabilities. The detection includes two steps: (1) first we need to identify the instructions that can potentially trigger vulnerabilities (*e.g.,* an arithmetic instruction may cause an integer overflow), this is done by various vulnerability detectors (*e.g.,* each detector is responsible for one type of vulnerability), and (2) to confirm the taint style vulnerability we need to decide whether any variable involved in the identified instruction is tainted by the user (*i.e.,* whether there exists a τ from $\mathbb{U}$ to the variable), SUTURE is able to detect the taint flow even if it crosses multiple entry function invocations, distinguishing it with previous works. In the remaining part of this section, we describe these two steps in detail.

Detector	Description
ITDUD	Tainted data usage in risky functions *e.g.,* `strcpy`, `sscanf`
TAD	Tainted data usage in arithmetic operations *e.g.,* integer overflow/underflow
TLBD	Tainted loop bound
TPDD	Tainted pointer dereference
TSD	Tainted size argument in data copy functions *e.g.,* `copy_from_user`

Table 3.1: Vulnerability Detectors used in SUTURE

Vulnerability Detectors. Dr.Checker has a collection of simple but well-defined vulnerability detectors, each of them targets instructions of a certain pattern (*e.g.,* conditional jump at the loop bound). Since our goal is to detect taint style vulnerabilities, we reuse Dr.Checker's detectors aiming at taint vulnerabilities (5 out of 8). We list our selected detectors and a brief description of their purposes in Table 3.1, more details can be found in the original paper [99]. We leave the development of more types of detectors as a future work. As mentioned above, SUTURE's novelty mainly lies in the ability to construct high-order taint flows, which is independent of the detectors.

Flow Constructor. The goal of the flow constructor is to enumerate all taint flows originating from $\mathbb{U}$ and sinking to a specified program variable, the existence of such taint flows means that the program variable can be controlled by the user. We sketch the algorithm for taint flow construction in Algorithm 1 while omitting some technical details like optimizations. As described in §3.3.3.8, the local taint flows are readily available in the taint summaries, in Algorithm 1 the function `getLocalTaintFlows` at line 16 returns all local taint flows sinking to the specified variable v. On the other hand, the function

Algorithm 1: Taint Flow Construction

```
1  Function getTaintFlows($trace$):
2      $f = \emptyset$;
3      $f_0 =$ getLocalTaintFlows($trace.top()$);
4      foreach $\tau_0 \in f_0$ do
5          if $source(\tau_0) \in \mathbb{U}$ then
6              $f.insert(\tau_0)$
7          else if $source(\tau_0) \notin trace$ then
8              $trace.push(source(\tau_0))$;
9              $f_1 =$ getTaintFlows($trace$);
10             $trace.pop()$;
11             $f = f \cup \{\tau_1 \circ \tau_0 | \forall \tau_1 \in f_1\}$;
12         end
13     end
14     return $f$;
15 End
16 Function getLocalTaintFlows($v$):
17     $s = \emptyset$;
18     foreach entry function $\varepsilon$ do
19         $s = s \cup \{\tau | \tau \in LT_\varepsilon, sink(\tau) == v\}$;
20     end
21     return $s$;
22 End
```

`getTaintFlows` tries to construct all taint flows (high-order or not), its argument *trace* is a variable stack that initially contains only the target sink variable. Basically, `getTaintFlows` constructs high-order taint flows by recursively connecting the local ones in a bottom-up style (*i.e.,* from the sink to the source), until a taint source variable in $\mathbb{U}$ is reached.

One thing to note is that a variable can be tainted multiple times during one entry function invocation and the later taint may overwrite the older ones. Since our static analysis is flow-sensitive, LT_{ε} contains local taint flows for all taint sites, however, when connecting the local taint flows to a high-order one, SUTURE will filter out those local taint flows whose taint will be overwritten at a later point, because currently we focus only on the sequential taint vulnerabilities (*e.g.,* we assume that different entry function executions cannot interleave), we leave the detection of concurrent vulnerabilities as a future work.

3.3.5 Output

If an user input tainted variable is detected in a sensitive instruction (defined by the vulnerability detector as mentioned in §3.3.4), a warning will be fired for the instruction. SUTURE assembles all the warnings in a program module as its output. Each warning specifies the warned instruction and its calling context, the warning type (*e.g.,* integer overflow), and the complete taint flow from the user input to the sink site, SUTURE also calculates and appends the *order* of each taint flow. To further help the reviewers screen the warnings, SUTURE also groups the closely related warnings together by their data flow relationship, regardless of the warning types (*e.g.,* a is the tainted variable in an overflow inducing instruction $c = a + b$, and c, which is tainted by a, is then used as a loop bound, the overflow warning and the loop bound warning will then be put into the same group), this way, the reviewers can avoid studying the common prefix of the taint flows repeatedly.

3.4 Implementation

As mentioned before, SUTURE is built on Dr.Checker, however, to improve the accuracy of the static analysis and support high-order taint flow construction, we re-write most parts of Dr.Checker's static analysis and implement many new functionalities (detailed in §3.3), in total, compared to Dr.Checker, SUTURE has 14482 LOC added and 2741 LOC removed in C++, plus 630 LOC of python scripts. In this section we discuss some implementation details of SUTURE.

LLVM Version. Dr.Checker is based on LLVM 3.8, which is too old to compile newer kernel versions nowadays. To test the latest kernels, we port Dr.Chekcer to LLVM 9.0.

Driver Module and Entry Function Identification. We follow Dr.Checker's approach [99] to identify the vendor driver modules and the entry functions for the evaluation, however, we update some out-of-date knowledge base (*e.g.,* structure definitions) used in Dr.Checker's scripts.

Dummy Object Creation and Matching. SUTURE supports arbitrary layer of memory indirection (*e.g.,* pointers), if a pointer derived from a global variable (*e.g.,* a pointer field in a global structure) is dereferenced but has no point-to records (*e.g.,* the setup code is out of scope of the program module), SUTURE will automatically create a dummy point-to object for the pointer and also set the object as a global taint source, this ensures that we do not miss any taint source in $\mathbb{G}$. Besides, to maintain the independence of the taint summary of each ε, the usage of a dummy object is refined within the analysis of only one ε (*e.g.,* each ε has its own dummy object copy for a same global pointer), while in the later vulnerability detection phase, the flow constructor will match multiple copies of a same dummy object to construct the high-order taint flows.

3.5 Limitations

We discuss the limitations of SUTURE in this section. On one hand, compared to Dr.Checker, SUTURE provides a more accurate static analysis and the capability to detect high-order taint style vulnerabilities, however, on the other hand, SUTURE also inherits some fundamental limitations from Dr.Checker.

Soundy but not Sound Analysis. Dr.Checker adopts a soundy but not sound static analysis (*e.g.,* skip analyzing the general kernel functions and limit the loop iteration times) for efficiency and accuracy [99], for the same goal, SUTURE reuses Dr.Checker's framework, thus its static analysis is also soundy. This means we may miss some vulnerabilities (*i.e.,* false negatives).

Unawareness of Sanitization Checks. Since SUTURE is not a fully path-sensitive analysis, it is unaware of some sanity checks for the tainted variables, which can cause false positive warnings (*e.g.,* though a tainted variable is used in a sensitive instruction, its value is already well limited by a sanity check). This is a well-known problem for path-insensitive static analysis, to address this issue, we can either extend our partially path-sensitive analysis framework to collect constraints for tainted variables, or run an independent symbolic execution pass to verify the warning, as done in some recent works [51, 130], we leave this as a future work.

3.6 Evaluation

3.6.1 Experiment Setup

Dataset. We pick one of the most complicated kernel drivers - the sound driver, from the latest Android kernel for Google's flagship mobile phone model Pixel 4 XL (the kernel version is 4.14, for Android 11) as the kernel module to test. We also prepare an older version of the same driver which contains two known high-order taint vulnerabilities as a ground truth, to test whether SUTURE can successfully detect them.

Hardware Configuration. We run the evaluation on a server with Intel Xeon E5-2695 v4 CPU @ 2.10GHz and 256 GB RAM, note that although the CPU has multiple cores, each SUTURE instance only uses one core.

3.6.2 Accuracy

First of all, SUTURE successfully detects the two known vulnerabilities [11, 12] from the older version of the sound driver, demonstrating its effectiveness in verifying the existing issues. Then for the latest version of the sound driver, SUTURE generates 360 high-order warning groups in total, out of which 82 are manually confirmed by us to be true positives, this yields a raw false positive ratio of 77.22% which is seemingly high. However, we want to argue that the raw FP ratio does not equal to the actually conceived FP ratio by the reviewers of the warnings, in fact, a large number of the false alarms share exactly the same root cause and as long as one such case has been inspected by the reviewer, the remaining can be easily and immediately filtered out (*e.g.,* in our results there are 205 false alarms featuring a same unique slice of instruction sequence in their taint flows, which can be manually identified from one of them and used to quickly filter out all other similar cases). Given this observation, we define the reviewer conceived FP ratio as N_{fp}/N_r, where N_{fp} is the number of false alarms and N_r is the number of warnings that are *actually* inspected by the reviewers (*e.g.,* in the aforementioned example although there are 205 false alarms, the reviewer only needs to inspect 1 of them), following this definition, SUTURE has a low reviewer conceived FP ratio of only 18.81%.

Reasons of FPs. We summarize the major causes of the false positives as following.

(1) Ignored Sanity Checks. As mentioned in §3.5, SUTURE is unaware of some existing sanity checks for the tainted variables, causing some false alarms. These sanity checks can be either explicit (*e.g.,* test the value range in a conditional statement, an invalid value may lead to an early termination of the program) or implicit (*e.g.,* the user is given only a limited freedom to select one value from a predefined set, which is secure). This is the most common cause of FPs for SUTURE, as well as many other bug finding tools based on static analysis.

(2) Infeasible Paths. Though SUTURE adopts a partially path-sensitive analysis to filter out many infeasible paths, the problem is still not completely solved - we can still find infeasible paths in the taint flows of the warnings, rendering them as false positives. More specifically, there are mainly two reasons for the infeasible paths: i) there exists false positives in SUTURE's indirect call resolution (*i.e.,* some recovered indirect call targets are actually

infeasible), and ii) there are some complex conflicting path constraints that are not captured by our partially path-sensitive analysis.

(3) Recursive Data Structures. The recursive data structure (*e.g.,* linked list) is another well-known difficulty for static analysis, since it is hard to statically decide the number of contained elements in such structures. In kernel, the most representative and widely-used recursive data structure is the linked list. To be conservative, SUTURE collapses the linked list to a single element (*i.e.,* it does not differentiate the elements in the linked list). Though a common practice, it can cause false alarms in some cases, for an example, a local taint flow τ_0 sinks to element 0 of a linked list while τ_1 sources from a different element 1, in theory, τ_0 cannot be connected to τ_1 since $sink(\tau_0) \neq source(\tau_1)$ (**Def 5** in §3.3.1), but SUTURE actually connects them due to the insensitivity to the linked list elements, resulting in false high-order taint flows. We leave the handling of recursive data structures as a future work.

3.6.3 Efficiency

SUTURE finishes the taint summary construction for the sound driver in 27.57 hrs, and the vulnerability detection phase in 2.35 hrs. The performance is significantly worse than the original Dr.Checker since SUTURE employs multiple enhancements (as detailed in §3.3) to improve the accuracy. Nevertheless, we believe the performance of SUTURE is acceptable given the capability to detect high-order taint vulnerabilities and the accuracy achieved, besides, sound driver is already one of the largest drivers in the kernel and we can naturally run multiple instances of SUTURE for multiple kernel modules concurrently, so it is well expected that we can finish analyzing more kernel modules (proportional to the CPU core count) within the same time frame with the prevalent multi-core CPU setup nowadays.

3.7 Related Work

Static Taint Analysis for Vulnerability Detection. There is a large amount of work applying static taint analysis on vulnerability detection, for different software written in different programming languages and at different layers. We discuss some significant categories as following.

For Android Applications. FlowDroid [45] is a widely-used high-precision static taint analysis designed specifically for Android applications, similarly, a large body of works utilize static taint analysis to detect information leakage of Android apps [69, 81].

For Web Applications. Many works focus on detecting taint style vulnerabilities in the web applications [85, 86, 48, 123, 62, 120, 116], which is very different from the main target of SUTURE (*i.e.,* the kernel written in C). However, it is worth noting that Dahse *et al.* [62] proposes a similar concept to high-order taint vulnerability - the second-order vulnerability in the web applications, where the taint flows via some persistent data stores (*e.g.,* database and files) on the server, compared to it, SUTURE analyzes the general global states in the program and supports constructing the taint flows of arbitrary order.

For Binaries. Cova *et al.* [60], DTaint [55] and Saluki [73] perform binary level static taint analysis based on symbolic execution to find vulnerabilities in the executable, while SUTURE's target is the open-source software, the availability of source code can benefit the analysis accuracy (*e.g.,* the exact structure layout), besides, symbolic execution based methods may not scale well for the large code base, especially when trying to hunt the high-order vulnerabilities.

For Open-Source C Programs. This is the most relevant category of SUTURE since we also target the C programs (*e.g.,* the Linux kernel) with the source code available. Chen *et al.* [53] uses static taint analysis to detect the implicit information leak in the network stack of the Linux kernel, Zhang *et al.* [131] applies the same technique to detect the uninitialized memory allocation in the kernel ION driver, KINT [122] utilizes static taint analysis to detect the integer errors in kernel and several user space programs. Dr.Checker [99] proposes a static taint analysis framework and supports to detect different types of taint vulnerabilities. Compared to the above works, SUTURE has multiple enhancements (§3.3.3) on the static analysis to make it more accurate, and more importantly, supports the detection of the high-order taint vulnerabilities. Yamaguchi *et al.* [128] and Shastry *et al.* [112] try to automatically infer the search patterns for taint vulnerabilities from known (in)secure code instances, which can then be used to detect relevant vulnerabilities, SUTURE on the other hand tries to find new taint vulnerabilities from the ground.

Improvements on Taint Analysis. Another category of related work centers on the improvements of the general taint analysis. P/Taint [75] tries to unify the taint analysis and point-to analysis to ease the implementation. Chua *et al.* [57] tries to automatically infer the taint propagation rules from the dynamic I/O analysis to avoid the inaccuracies of the human defined rules, based on the similar insight, Neutaint [113] employs neural network to conduct the dynamic taint analysis. Heo *et al.* [79] uses machine learning to guide the switch between sound and unsound static analysis, taking the best of both worlds.

We consider these works as complementary to SUTURE, since they can potentially help SUTURE's static analysis to be more accurate or efficient.

3.8 Conclusion

In this chapter we introduce SUTURE, a highly accurate static point-to and taint analysis system for the kernel that is capable of constructing high-order taint flows in an efficient way, this enables it to effectively detect the stealthy high-order taint style vulnerabilities in the kernel modules. The evaluation result shows that SUTURE can both confirm known high-order vulnerabilities and successfully find new ones in the kernel, with a low conceived false positive ratio for the warning reviewers, and an acceptable performance.

Chapter 4

Study Attacks Against Deployed Kernels: The Case of One-Click Root Apps

4.1 Introduction

We are in an age when customers are not given full control over the purchased personal mobile devices such as smartphones and tablets. Due to the popular demand by users, a unique ecosystem of offering smartphone root or jailbreak has formed. Root and jailbreak are the process of obtaining full privilege on Android and iOS devices respectively. They allow users to bypass restrictions set by carriers, operating systems, and hardware manufactures. With full control over the device, a user can uninstall bloatware, enjoy the additional functionalities by specialized apps that require root privileges, or run paid apps for free.

Classified by whether a device is flashed, there are two types of root methods: 1) soft root. 2) hard root. The former refers to the case where root is obtained directly by running a piece of software (*i.e.,* root exploits). The latter refers to the case where *su* binary is flashed externally via an update package or ROM. Depending on the device model and OS version, different root methods may be applicable. For instance, due to locked bootloaders, some devices cannot use hard root. Similarly, if a particular device has no software or hardware vulnerabilities whatsoever, soft root would not be possible. In practice, like any other systems, Android devices do have a variety of vulnerabilities in various components: kernel, driver, and application as summarized in §4.2.

In this chapter, we focus on the soft root as the same exploits can be potentially abused by malware authors and therefore much more dangerous than hard root. In Android, such root service is provided by a number of parties. Individual developers or hackers often identify vulnerabilities, develop, and publish exploit tools to gain fame and possibly fortunate. However, due to the diversity of Android devices in terms of hardware, fragmented OS versions, and vendor customization [103], it is simply not scalable for individuals to engineer a large number of exploits to cover a wide range of the devices. Therefore, the business of offering root as a service has emerged [24, 20, 27].

Interestingly, most commercial root providers are free to use. They operate by requiring the exploit to run on an Android device by a user voluntarily, *e.g.,* through an one-click root app [24, 20, 27]. Unfortunately, attackers can also acquire such exploits easily by impersonating a regular user. To make the problem worse, some of the large root providers have a large repository of root exploits or even invent new ones so they stay ahead of their competitors. This may give attackers a strong incentive to target such providers.

In this chapter, we examine the root ecosystem closely to understand the following high-level questions: (1) How many types and variations of Android root exploits exist publicly and how they differ from the ones in commercial root providers. (2) How difficult is it to abuse the exploits offered by the root providers. We answer the above questions by undertaking a series of measurement and characterization of root exploits as well as the providers that offer them.

The contributions of this chapter are the following:

- We conduct a comprehensive measurement study on Android soft root methods to understand their origin and overall trend. We find that 1) most public Android root exploits target the application-layer vulnerabilities that affect only specific types of devices. 2) Although kernel vulnerabilities are considered the most dangerous, an exploit developed on one device may need to be adapted to work on another. 3) As kernel vulnerabilities become rare, device drivers become the dominating target to find root exploits.
- We analyze the security protections employed by a number of root providers on their exploits. While larger root providers often employ more protections, we identify systematic weaknesses and flaws which substantially undermine their effort. The result calls for better security practices on protecting such dangerous exploits.
- We survey the availability and variety of the exploits online versus the ones extracted from root providers which range from large security companies to individual devel-

opers. We report that a large root provider not only keeps “secret” exploits, but also spent significant engineering effort to polish and adapt existing exploits.

4.2 Publicly Available Android Root Exploits

In this section, we attempt to exhaustively collect all publicly known Android root exploits or vulnerabilities and understand their characteristics. Even though root exploits are reported to have been used by malware in the wild already [136, 76, 37], we still lack a complete and up-to-date picture. We are not aware of any systematic research studies on root exploits used by malware in the recent two years. It is unclear what exploits may be currently used and which ones are easily usable by malware, thus likely to appear in the future. We aim to understand the question by analyzing the current and publicly available resources to malware authors.

Data sources and collection methodology. We survey a large number of public sources including academic papers [78], research projects [5], published books [65], as well as online knowledge base(*e.g.,* CVE database or forum post such as XDA forum) [21, 16, 7, 15]. Search terms including “Android root”, “root exploit”, and “privilege escalation” are used to locate the relevant information. Note that even though we attempt to collect an exhaustive list based on our expertise, it is inherently a best effort. The list eventually leads to a dataset of 73 exploits or vulnerabilities.

In most cases, a vulnerability (with a CVE number) maps to a corresponding exploit. However, as will be described in §4.2.2, we are unable to locate publicly available exploits for some small subset of CVEs. In many other cases, the opposite is also possible —

no CVE number is assigned but the exploit is readily available, likely because of its limited impact on very few device types.

We also observe that some exploits require multiple vulnerabilities to gain root. For instance, master key vulnerability (*e.g.,* ANDROID-8219321) only leads to system user privilege. Additional vulnerabilities are necessary to complete a root exploit [30]. In such few cases, we consider them two separate ones. In the survey, we found 5 vulnerabilities which can gain system privilege only and 3 vulnerabilities which can gain root permission from system privilege (we count them as 8 still). In addition, some exploits are related to each other and can be considered variations of one another. When it is possible to locate different CVE numbers or vulnerabilities through the technical details, we also consider them different exploits. This inclusive strategy is more likely to lead to a more complete discovery of exploits and vulnerabilities.

At a high level, we are able to locate enough technical descriptions of the vulnerabilities or exploits, although they vary significantly in detail, clarity, and availability of source code or binaries. Perhaps expectedly, we find that the majority of the exploits are not produced by academic research.

Questions to answer. We aim to answer the following questions from the analysis:

(1) How many general vs. specific exploits exist? Intuitively, some exploits are more general than others, especially those exploiting kernel vulnerabilities. Some exploits may be applicable to certain vendors only.

(2) Whether the exploit source code or binaries are publicly available? What's the requirement to run the exploit? Can the exploit work via a standalone app, *e.g.,* without user intervention or booting into recovery mode?
(3) Whether antivirus can recognize the exploits?

4.2.1 Root Exploit Impact and Coverage

To understand the impact and coverage of an exploit, we first try to identify the layer that is targeted. This is because the impact could be different depending on the layer. For instance, if it is a vulnerable setuid program (in the application layer) installed only a certain models of a vendor, then its impact will be limited. We divide the layers into four categories based on the Android Architecture:

Linux Kernel. Due to its privileged position, targeting Linux Kernel is natural to achieve full control over an Android device. In particular, a vulnerability in the kernel has a large impact as all devices that run the vulnerable kernel can potentially be affected. For instance, TowelRoot (CVE-2014-3153) exploits the *futex* syscall bugs to gain root access and it is considered to affect all kernel versions before 3.14.5. In this category, we include everything running inside kernel except the cases described below.

Vendor-Specific Kernel or Drivers. Different from the main kernel code that runs on almost every device, vendors either customize the kernel (*e.g.,* Qualcomm's custom Linux kernel branch) or provide vendor-specific device drivers for various peripherals (*e.g.,* camera, sound) [135], Such code runs inside the kernel space and the compromise of which can also lead to full control over the device. Given that they are produced by a single party without much open auditing, and sometimes closed source (*e.g.,* especially the device drivers), the chance of them having security vulnerabilities can be high, as confirmed in our measurement results. However, since not all Android devices run the same set of customized kernel or device drivers, an exploit on a specific customized device can only impact a subset of Android devices (*e.g.,* certain Samsung devices).

Libraries Layer. Exploits at the libraries layer target the Android libraries or external libraries used for supporting different applications. For instance, in ZergRush exploit (CVE-2011-3874), libsysutils used by Volume Manager daemon (running as root) in Android is shown to have a stack overflow vulnerability that leads to root privilege escalation [65]. The vulnerability in such libraries can have a large impact because they may be embedded by multiple programs, as long as one such program runs with root privilege and exercise the vulnerable code, a root exploit can be successfully constructed. The ObjectInputStream vulnerability (CVE-2014-7911) is another example.

Application and Application Framework Application layer root exploits mostly include vulnerable logics introduced by setuid utilities, system applications, or services. The impact of such exploits depends on whether it is a third-party one or not. So far, most cases are from third-parties which indicate a limited impact. One example is a vulnerable setuid utility that is only present on XoomFE devices that has a command injection vulnerability [39]. Another instance is a backdoor-like setuid binary shipped with certain ZTE Android devices (CVE-2012-2949).

In general, from the highest to lowest, the order of impact and generality of exploits would be 1) the kernel exploits, 2) the exploits targeting libraries that are used by Android

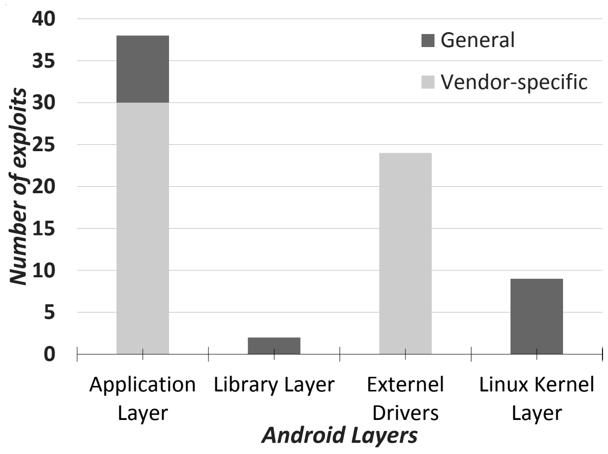


Figure 4.1: Number of exploits by layer

system processes, 3) exploits targeting system applications or services, and 4) exploits against vendor-specific device drivers, applications, and programs.

Even though we cannot accurately predict the number of devices impacted by each exploit, the reasoning is that kernel and Android system code is much more widely used than the vendor-specific code. In addition, patches of kernel vulnerabilities are much harder to reach the end-user whereas application updates can be quickly pushed out.

Breakdown by layer. As shown in Figure 4.1, out of 73 exploits, there are 54 exploits that are vendor-specific (in lighter gray) and 19 general exploits (in darker gray). The vendor-specific ones include all device driver exploits, and vendor-specific applications or programs. The *application layer* is found to have the largest number of exploits, although most of them are vendor-specific. The *external drivers layer* has the second largest number of exploits but all of them are vendor-specific by definition. It is expected that the *kernel layer* and *library layer* have the smaller number of exploits which are very general and

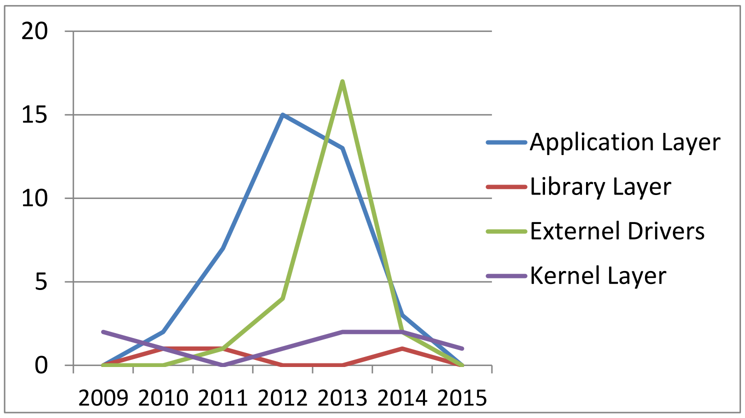


Figure 4.2: Number of exploits by year

extremely dangerous. The number of new *kernel layer* exploits occurred each year is also relatively stable in our survey – only one or two per year on average.

Time dimension. Another important dimension is time. Specifically, the lifetime of vulnerability is determined by the patch version and date. The later the discovery, the longer the vulnerability lives and therefore a higher impact. On the other hand, the sooner the discovery, the more quickly the root exploits can be developed. Figure 4.2 shows the number of exploits discovered in each year. As we can see, vulnerabilities explode around year 2013 due to a large number of vulnerabilities introduced by vendor customization at the *external drivers layer* and *application layer*. One of the key problems is that device files on many vendor-customized Android systems have weak permissions [135], without which an app process cannot even open these device files and launch exploits. Such period also coincides with the increased market share of Android and participating vendors. On the other hand, the kernel and library layers have a relatively stable pattern.

Obviously, with common mistakes corrected in the vendor customization process, the number of vendor-specific vulnerabilities will drop. However, it is always hard to predict the new trend or classes of exploits that may surface. As long as there is strong need from users, we believe root exploits will still continue to exist in the foreseeable future. For instance, at the time of writing, a new kernel-level root exploit named PingPong root [126] is announced.

Coverage. Theoretically, a kernel vulnerability affects all kernel versions between when the vulnerability is introduced and when it is fixed. Therefore, a recently discovered kernel vulnerability such as TowelRoot (CVE-2014-3153) should have a significant coverage. However, as will be discussed in §4.3, it is most often not the case. In practice, a kernel exploit may depend on system configurations, address space layout, compiler options, *etc.*. Therefore, to successfully root a device, multiple exploits are usually attempted in both the malware [136] and the root providers.

4.2.2 Exploit (Source or Binary) Availability

In this section, we aim to understand how readily available the exploit source code or binaries are on the Internet for public use. In particular, the availability is a direct indication on whether malware authors can find and leverage such exploits. Even though it is well known that malware already start to embed root exploits that are often copied from the public sources [136], it is unclear how many such exploits can be located and abused.

To locate exploit source code or binaries, the methodology is to simply use the relevant keywords (when applicable) of an exploit that typically include the CVE number (*e.g.,* CVE-2014-3153), the Google Bug ID (*e.g.,* ANDROID 3176774), impacted device model, and the exploit nickname (*e.g.,* TowelRoot). To ensure adequate coverage, we undergo two rounds of independent web searches.

Out of the 73 cases, we are able to locate either the source code or binary of 68 exploits. Only 5 of them have neither found. One of them is not available because it is only described in a research abstract. Others are not available even though the corresponding CVEs clearly indicate they allow arbitrary code execution with elevated privileges. We are not certain about the root cause but one plausible explanation is that the person who discovered the vulnerability did not release the technical details or any proof-of-concept exploits. It is also possible that the vulnerabilities are not generic enough to attract individual hackers to build an exploit.

Theoretically, both the binary and the source code are valuable to malware authors. A malware can embed the binary directly so long as it is an independent piece (*e.g.,* executable or libraries) and has an easy-to-identify interface. Of course, source code has many advantages since it can be freely customized and improved.

Overall, there are 46 exploits with source code available, 18 of them are simple exploits that leverage weak file permissions and symbolic link attacks [28] which are typically introduced by vendor customization. Such exploits can be mostly implemented in shell scripts. The rest are written in C (and one in Java against CVE-2014-7911). On the other hand, there are 22 exploits with binaries available only, which are in the following two forms: 1) PC-side scripts that may push additional binaries onto the device and 2) apk files that run on the device directly. There are 10 and 12 of them respectively.

We observe that even though source code is generally more valuable, it may not be as robust as the binaries, especially when the source code is offered as "third-party" proof-of-concept. Particularly, in order to accommodate different devices and models, considerable iterations and engineering efforts are required. For instance, TowelRoot is binary only and it has evolved over three major revisions supporting different devices. The available source code, however, is just proof-of-concept and is written by other developers [10].

To summarize, malware will likely be able to integrate most of the exploits, even if some may have limited coverage.

4.2.3 Exploit Requirements

Even if an exploit source or binary can be found, one still needs to understand the requirement to run them. For instance, an exploit may require an adb shell setup through a PC connection — since only processes running as *shell* user can perform the exploit. In other cases, an exploit may require user interactions (*e.g.,* booting into recovery mode at least once to trigger the vulnerable code).

To understand the exploit requirements, we perform two steps: 1) locate technical reports or tutorials published either by the exploit authors or other interested parties [39]. 2) if 1) is not available, we attempt to read the exploit source code or script, which will typically contain such information. It turns out the two steps can cover most exploits.

From the technical details of the exploits and source code, we are able to identify the following major requirements (from the most rigid to the least):

- **Requiring user interactions.** This category have few cases. One case is asking the user to download an app and manually interrupt the installation [18]. One is asking the user to boot into recovery at least once [34]. Another is asking the user to manually put the device into "battery saving" mode [26]. The last asks the user to open a vendor-specific app and hit a button [29]. Intuitively, exploits in this category are difficult to be used by malware authors to fully automate the exploit.

- **Requiring adb shell through a PC connection.** For some exploits, adb shell connection is required because of the following most common reasons: 1) The exploit can successfully modify a setting in local.prop which enables root for adb shell only. 2) The exploit needs to write to a file owned by group *shell* and group-writable (not world-writable) [28]. 3) The exploit targets the adb daemon process that requires the attack process to run with *shell* user. For instance, the Rage Against the Cage exploit [25] targets the vulnerability of adb daemon's missing check on return value of setuid().

- **Reboot.** Generally, many root exploits require at least one reboot. For instance, a symbolic link attack would allow an attacker to delete a file owned by *system* with weak permission, *e.g., /data/sensors/AMI304_Config.ini*, to setup a link at the same location to a protected file, *e.g., /data/local.prop.* After a reboot, the corresponding init scripts would attempt to change the permission of the original file (*i.e., /data/sensors/AMI304_Config.ini*) to world-writable, which in reality changes the permission of the linked file (*i.e., /data/local.prop*).

- **None or permission.** The exploits in this category have no hard requirements, however, some of them may require certain Android permissions like *READ_LOGS* in order for the process owner to be placed in certain user group.

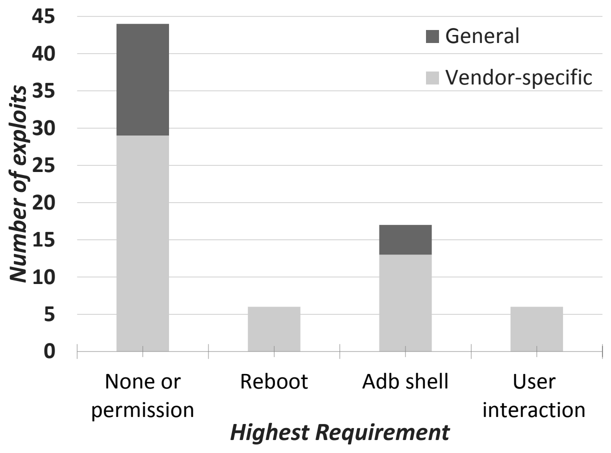


Figure 4.3: Exploit requirement breakdown

If an exploit has multiple requirements, we will only count the most rigid one (*e.g.*, one exploit needs both adb shell and reboot will fall into "adb shell" category). The results are summarized in Figure 4.3. 6 of them require user interactions. 17 of them require adb shell through a PC connection. 6 of them require rebooting the device. The rest 44 do not have any hard requirements, in which 5 do require certain Android permissions. Most exploits are vendor-specific, however, there are 4 and 15 general exploits in "Adb shell" and "None or permission" categories, respectively.

Correlating with the 68 exploits that have source code or binaries, there are 39 available exploits that need only reboot, permission, or none and can potentially be abused in a malware silently gaining root access.

4.2.4 Root Exploits Detection by Anti-Virus

Since root exploits can be potentially abused by malware to gain the highest privilege, we expect they are of high priority to antivirus software. To verify the hypothesis,

Root exploit	AVG	Lookout	Norton	Trend Micro
exploid(2010)				
Zimperlich(2010)	X	X		
Gingerbreak(2011)	X	X	X	X
BurritoRoot(2012)	X	X		X
Poot(2013)				X
LGPwn(2013)			X	X
WeakSauce(2014)	X	X		
Framaroot(2014)	X			X
Towelroot(2014)	X	X	X	X
PingPongRoot(2015)				X

Table 4.1: Detection results of mobile antivirus

we download 21 root exploits in the form of 10 apk files or ARM ELF executables. In order for the antivirus to recognize a malicious app, when there is a lack of apk file, we package the ARM ELF executables into a simple Android app (stored in the libs folder). Since some source code is only proof-of-concept, we decide to use the binary only for experiment.

We downloaded 4 antivirus software from the Google Play: AVG AntiVirus Free 4.3.1.1.213361, Lookout Security and Antivirus (lookout) 9.18.1, Norton Mobile Security V3.10.0.2361, Trend Micro Mobile Security V6.0.1.2050.

We use a Galaxy S3 phone to carry out the experiment in the early May of 2015. We install the antivirus software one at a time and never keep two or more running simultaneously to prevent potential conflicts. The antivirus software all have real-time protection enabled and will pop up a window when they believe a malware or suspicious app is detected.

Table 4.1 shows the results with exploits ordered by year. Note that in the case of Framaroot, 12 exploits are packed in the same apk, and only one row is shown to represent them. As we can see, most exploit binaries are flagged by more than one antivirus software, which is expected as most of them are well-known root exploits.

There is one exploit, exploid that cannot be recognized. Poot and PingPong root are detected by one antivirus only, indicating that some exploits can still fly under the radar (The reason may be that our samples are different from those used by malware). Interestingly, the very recent PingPong root exploit, published a few days ago at the time of writing (early May 2015), is already detected by Trend Micro, indicating that they are specifically sensitive about the publicly available exploits.

In contrast, as will be shown in §4.6.3, the exploit binaries engineered by large root providers are surprisingly "clean" as all major antivirus software have difficulty detecting them.

4.3 Adaptation of Root Exploits

Android is well known for its fragmentation due to carrier and vendor customization [103]. On one hand, the availability of a large number of customized Android devices allow greater market penetration. On the other hand, the diversity of Android devices makes it extremely difficult to write robust root exploits that work on devices with varying application/kernel configurations and settings.

It is known that many exploits require adaptations to work across devices. In fact, it is believed that adaptations directly discourage the malware authors to use certain exploits (*e.g.,* zergrush and mempodroid) in the wild [76].

Typically, an exploit needs specific environment to work. Difference in CPU, kernel version, OS version may cause a failure. In memory corruption based exploits, requiring the knowledge of absolute or relative memory addresses of certain key data structures are common reasons why adaptation may be necessary. For instance, in the exploits against CVE-2014-4322, the address of a kernel symbol needs to be determined which differs across devices. Note that a bruteforce approach will not work as it may crash the system jumping to an undesirable address. In rare cases such as Exynos-abuse, adaptations may not be necessary. This exploit can access arbitrary physical memory through a vulnerable device driver and overwrite kernel data to gain root privilege. Instead of using a hard-coded static address, the exploit can search from the beginning of kernel space to locate the target kernel symbol. Of course, such a special vulnerability allowing the whole kernel space access is unlikely to occur and is limited to certain device type only. In general, kernel exploits require substantial adaptation, as is demonstrated through the case study below.

4.3.1 CVE-2014-3153 (Futex bug)

The futex kernel vulnerability is reported to affect all kernel versions prior to Jun 3rd 2014 and its first exploited by TowelRoot. It was originally designed to root Verizon Galaxy S5, then modified to be compatible with more devices, including ATT Galaxy S5,Galaxy S4 Active, Nexus 5. Although it claims to possibly work on every android device with a vulnerable kernel, a slight variation of hardware platform or kernel versions may cause this exploit with high precision requirement to fail. To cover more devices, it adds a

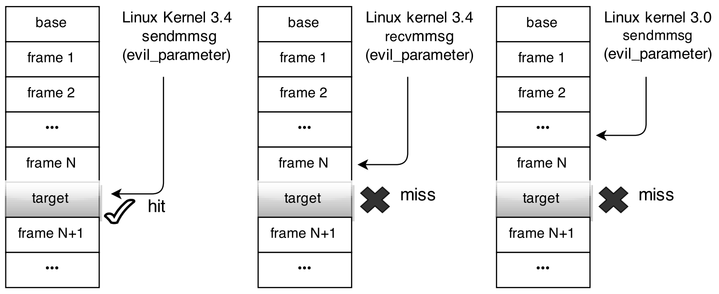


Figure 4.4: CVE-2014-3153: kernel stack overwrite by invoking system calls

feature named mod strings for users to modify 5 different exploit variables. We explain one of the key variables, system call, in detail below.

The system call variable specifies one of the four possible system calls utilizable to carry out part of the exploit. The context is that the attack sets up a pointer in kernel heap to point to a kernel stack address that is subject to overwrite by a system call. As shown in Figure 4.4, an attacker needs to pass a malicious parameter to a system call, which will be copied into the kernel stack, and hope that it will eventually land on the target address in kernel stack. Depending on the exact kernel version and configuration, there are two obstacles: 1) the target stack address may be different relative to the base stack address. 2) the depth that the malicious system call parameter reaches can also be different. We illustrate such obstacles in the figure. In the first case, the parameter of syscall `sendmmsg()` can be successfully placed to overwrite the target stack address. In the second case, however, due to the wrong syscall chosen, the parameter may fail to hit the target address. In the third case, the same syscall `sendmmsg()` is chosen, but due to kernel version difference, the reached depth is different, missing the target. Due to the above difficulties,

Name	Components	Devices supported (claimed)
Root Genius	PC/MOBI	20,000+
360 Root	PC/MOBI	20,000+
IRoot	PC/MOBI	10,000+
King Root	PC/MOBI	10,000+
SRSRoot	PC	7,000+
Baidu Root	PC/MOBI	6,000+
Root Master	PC/MOBI	5,000+
Towelroot	MOBI	N/A
Framaroot	MOBI	N/A

* **PC**:PC-side software **MOBI**:Android app

Table 4.2: Root Providers List

Towelroot suggests users try different syscalls in combination with other variables to hope that the target address will be hit.

In practice, we have tried the Towelroot on 3 devices that we can get our hands on, all with a vulnerable kernel built prior to Jun 3rd, 2014, yet Towelroot fails to root 2 of them.

4.4 Root Providers Overview

As alluded before, there exist a large number of root providers, ranging from ones developed by individuals to large companies. In this section, we aim to study them in the following aspects: 1) survey different types of root providers and understand how they operate. 2) characterize the protection strength on the carried dangerous root exploits. 3) measure the extracted providers' exploits and understand their relationship with the publicly available ones.

In general, the discrepancy of the available resources between small and large providers is likely a key factor in deciding the above aspects. There are a number of popular root providers which contain both the largest and newest ones as listed in Table 4.2. We confirm that the larger providers do offer a much more comprehensive set of exploits, however, even though present, the corresponding protections are substantially far from being adequate. In fact, we find serious weaknesses that allow us to extract and study a large portion of the exploit binaries from one large provider.

We study 7 out of the 9 providers in depth and anonymize their names for security concerns, as shown in Table 4.3.

Methodology and collected results. We collect information of three main categories: 1) Public information about each provider, *e.g.,* number of devices that they claim to support, whether it has a PC-side program and/or an independent Android app, as shown in Table 4.2. 2) Exploit information including the location of the binaries (*e.g.,* on a remote backend or local), and the quantity of them. 3) Protection employed by root providers to prevent the exploit binaries from being reverse engineered and abused by others. This gives a rough estimate on the level of difficulty to extract the valuable exploits for malicious purposes. The information collected in 2) and 3) requires understanding of the inner workings of providers via reverse engineering.

Root provider architecture. From all the providers we studied, a common architecture is depicted in Figure 4.5. The service is typically through either a PC-side program and/or an independent Android one-click root app. The former can control a device via the ADB interface and thus utilize both ADB-related exploits like "rage against the cage" and others

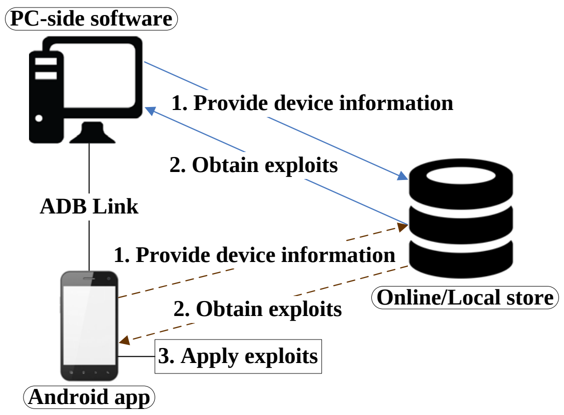


Figure 4.5: General Root Architecture

launched directly on the Android device. The latter Android app can operate independently to execute root exploits.

The main program logic involves three key steps:

STEP 1: Collect device information such as model name, kernel and Android version, hardware platform and so on.

STEP 2: Based on the information, obtain proper exploits from either remote servers or local store.

STEP 3: Execute the chosen exploits on the device to gain root permission.

As shown in Table 4.2, the providers are sorted by their coverage. It is obvious that all larger root providers are comprehensive in offering both the PC-side program and the independent Android app, whereas smaller providers typically offer exploits in one way or the other.

Number of exploits. Here we focus on the exploits that can be launched directly through the Android app, since they are much more likely to be abused once stolen by attackers. In

NO.	Exploits		Protection			
	Store	Amount	Store	PC-side	Device-side	Exploits
1	LOC/OL	160+	OL	A	ANP*	COPS
2	LOC/OL	60+	OL/LOC	N/A	NO	CS
3	LOC	40+	LOC	None	ANP	S
4	LOC	20+	LOC	N/A	O	CS
5	LOC	20+	None	P	N/A	None
6	LOC	10+	None	N/A	None	C
7	LOC	10-	None	N/A	None	O

*: Not true for its app from a special channel
OL : Online **LOC** : Local **N/A** : Not applicable or studied
A : Anti-debug **C** : Tamper-detection **N** : Code Protection with JNI
O : Obfuscation **P** : Packing **S** : String Encryption

Table 4.3: Root Providers Measurement Result

Table 4.3, we list the number of exploit binaries we are able to locate for each provider. It is surprising to see that the number goes over one hundred for the largest one. In addition, the number is only a lower bound as there can be others that we are not able to find (See §4.5.1 for details on how to locate the exploits). Such number is significantly higher than what we can find from public sources, highlighting the potential risk of being targeted by attackers. Note that we sort the table by the number of exploits we can find, yet it does not correspond to the same order presented in Table 4.2, therefore, not revealing the provider names.

We do realize that different providers may organize their exploits differently into binaries. One binary could correspond to a single exploit with or without its variants. Therefore, simply counting the number of exploit binaries can be biased. In §4.6, we offer a more comprehensive analysis on the exploits and compare them with the publicly available ones.

Protection Strength. Perhaps expectedly, we observe that larger root providers with more exploits tend to employ stronger protection of their products, and smaller providers usually employ little to no protection. For instance, as shown in Table 4.3, provider 1 and 2 not only protect the Android one-click root apps, but also introduce tamper-detection and encryption in their exploit binaries (typically in native code) to prevent them from being stolen and abused. In addition, the network communication to retrieve exploits from its remote store is also encrypted. In contrast, provider 6 and 7 only equip some basic protection in their exploits, which is easy to bypass. In the study we also find that some larger providers will integrate small providers' apps or exploit binaries directly, this observation again reflects the lack of protection for small root providers which are usually individual hobbyists. Unfortunately, as we will show in §4.5.1, the seemingly strong protections in large providers can in fact be broken down fairly quickly due to several severe weaknesses we identify.

PC-side vs Device-side Protection. It is important to realize the PC-side program and independent Android app contain duplicate functionality of reading device information and retrieving exploits from local or remote store. Therefore, as long as either one has a weak protection, the procedure can be revealed and exploits maliciously retrieved. Indeed, security is only as strong as its weakest link.

Interestingly, we observe the protection strength is indeed typically inconsistent. Compared to PC, Android has a much shorter history which results in fewer available commercial-grade protection methods, *e.g.,* VM-based protection [46]. This is supported by our finding that most providers have weaker protections on its Android app compared to its PC counterparts, even though they usually throw in a number of protections, hoping that they are strong enough (*e.g.,* ProGuard). On the other hand, provider 3 does employ a stronger commercial protection solution called Bangcle on its Android app, yet it has no protection whatsoever on its PC program. The result is summarized in Table 4.3. In the cases of a "N/A", it indicates that we did not study it since the other side can already be successfully reverse engineered.

Besides above observation, extra opportunities exist where inconsistently protected software may be distributed. First, older versions and newer versions of the same software may implement the same core functionality, but stronger protections are added only to the newer versions. For instance, we observe an old version of an Android one-click root app from a provider has significantly weaker protection than the new ones. Second, in rare cases, some root providers may share code with each other, yet one version may have much weaker protection than others. We observe one such case among the providers - the involved two providers have a cooperating relationship.

4.5 Case Studies of Protection Mechanisms

Given the competitiveness of the providers is purely determined by the variety and quality of the engineered exploits, they should be highly security minded, whether individual hackers or large companies who even offer security products. We expect to see best practices in protecting their code. However, even when strong protection is indeed employed, we identify some critical (and some obvious) flaws which greatly undermine the effort. In the end, we are able to seize virtually all exploit binaries offered by the root providers. In this section, based on the number of exploits, we divide the providers into three categories. From each category we will choose representative providers for detailed study, aiming to locate flaws and weaknesses in their protection methods.

As depicted in Section §4.4, there are three steps in the root procedure. By reversing how each step is performed, one can easily steal all exploit files and run them in any piece of malware to gain root privilege. Even the most difficult provider only took a graduate student, who is not a professional hacker, about one month of part-time work to finish, which is far less than expected for such a highly sensitive service. For reference, it took a professional Symantec research group about six months to figure out the basic structure and behavior of Stuxnet [66], which is a piece of state-sponsored malware created to attack industrial control systems. In the rest of the section, we will describe the protection methods and the weaknesses in each step for different root providers in detail.

4.5.1 Large Root Providers

Provider P1 (we will refer provider n in Table 4.3 to Pn) is one of the largest root providers currently with over a hundred exploit binaries. Its service is provided by either the PC-side program or the Android app. The most critical part in P1's architecture is an online exploit store. To update the service, P1 simply needs to add new exploits to the store. For a given device, only a selected subset is downloaded and attempted.

Protection Methods.

STEP 1: Provide device information. P1 encrypts the gathered device information such as Android device model and kernel version before sending them out to a remote server with a combination of standard encryption algorithms. Similar protection is also widely used by other large root providers to secure their online exploit stores.

STEP 2: Obtain exploits. After receiving encrypted device information, P1 servers first return a file which is an array of exploit descriptors. Each descriptor contains elaborate information about a specific exploit including an internal exploit identifier, a download link, and comments such as the affected devices. Related exploit binaries can then be fetched based on its descriptor. The descriptor file is encrypted with the same algorithm as in *STEP 1*. Besides, each file URL is encoded in a random string to prevent exhaustive crawling. A similar "descriptor file mechanism" can also be observed for P2, but with a different format.

STEP 3: Apply exploits. P1 encapsulates each exploit into a separate Linux dynamically linked shared object file (.so). These library files share a common interface of entry point and thus can be executed, in a uniform fashion, one after another. Such files are downloaded every time when the PC-side program or the Android app is run. It is obvious that such files have to be protected in order to prevent misuse. We encounter the following: 1) The code is obfuscated by redundant instructions [96] and a custom re-arrangement procedure of the ELF binary to destroy the header and prevent disassembling. 2) A custom packing method scrambles the actual exploit code. 3) Most constant strings are encrypted. 4) There is a tamper-detection in every exploit file to ensure that the exploit can only be launched by an authentic P1 product (its own Android app), based on the app's embedded signature or the package name.

Security Flaws. Unfortunately, there exist a number of flaws that substantially undermine the strength of the protection employed by P1. We highlight them below:

- *Inconsistent protection for the same Android app obtained through different channels.* After studying P1 for a while, we realize that there are in fact two ways to get its Android app: 1) Download from P1 official website or other third-party app markets directly (Google Play prohibits such apps to be published). 2) Obtain the copy from the PC-side program's download cache. This is possible since the PC-side program will download and install the app to the connected device automatically if none is detected.

Surprisingly, the apps from these two channels behave exactly the same on mobile devices, yet there is a world of difference in their protections. The one downloaded from the official website is well protected with main "Classes.dex" encrypted and packed. This is an effective practice found in some commercial solutions (*e.g.,* Bangcle). The one obtained through the PC-side program, in contrast, does not include any protection whatsoever. Considering that Android apps tend to update frequently with only minor changes, if the core encryption logic remains the same in future versions, an attacker can misuse it for a long period of time to continually extract new exploits developed by the provider. This flaw effectively reveals all encryption algorithms used in *STEP 1* and *STEP 2*.

- *Custom obfuscation procedure leaked through online security contest held by the provider itself.* P1 employs some obfuscation methods such as a custom redundant instruction pattern and ELF header scrambling, these methods are in fact exposed in an online security contest. By simply reading the answers provided by the crowd, all details are revealed, including the obfuscation pattern and the way to restore metadata in the ELF header. Once the obfuscation is understood, remaining protections in *STEP 3* are much less effective.

- *Discrepancy in protection strength of device-side and PC-side software.* Similar to what is discussed in §4.4, the unprotected Android app of P1 obviate the need to deal with the PC-side program protection such as anti-debug. The opposite occurs in P3 where unprotected PC-side program enables us to ignore its well-protected Android app.
- *Leave informative names of critical functions untransformed.* Root providers often employ standard cryptographic and compression algorithms (*e.g.,* AES) to protect the code and data. However, if such obfuscation logic leaves its function and variable names untransformed (*e.g.,* a function named "md5" or a variable named "AESKey"), one can immediately recognize the algorithm and reverse the obfuscation. Such form of leakage exists in both SMALI and ARM native code of P1 and many other root providers, which undermines their protection drastically. This flaw impacts *all three steps* of P1.
- *Vulnerable tamper-detection mechanism* Signature or package name based tamper-detection can be found in many providers' exploit files. However, the detection is executed only one time at the beginning, which makes it easy to bypass — modifying one conditional jump suffices and works in all cases. Scattered and repeated tamper detection will substantially raise the protection level in *STEP 3.*

To verify that all P1's protections are successfully bypassed, we develop a piece of proof-of-concept Android malware which can fetch and run the root exploits as well as successfully obtaining root privilege on a few tested devices, including HTC One V and Sony Ericsson ST18i. In theory, this malware can leverage the full capacity of P1 since it can use all current and future exploits P1 maintains, as long as the procedure remains the same. Although we did not include the exploits that can only be launched from the PC program, they can also be downloaded and used the same way.

4.5.2 Medium Root Providers

We choose P4, a popular moderate-sized provider to study in this section. Different from P1, P4 stores all its exploits locally. Although there are some protection for the local store, it is overall much weaker than P1's protection. It only took us three days to obtain all P4's exploits and bypass the protection mechanism, which will be described below.

Protection Methods

STEP 1: Provide device information. Since all P4's exploits are stored locally, there is no need to send device information to any remote server. All device information is gathered locally and will then be used to guide the selection of proper exploits.

STEP 2: Obtain exploits. As soon as a specific exploit is considered proper for current device, P4 will fetch it from the local store. There are two layers of protection in this process: First, inside the Android app, MD5-based name transformation procedure is used to map an internal exploit name to the corresponding obscured file name in local store. Second, actual exploit binaries are compressed in gzip, while no informative file suffix can be seen.

STEP 3: Apply exploits. P4's exploits are all ELF executables. Similar to P1, there is also a package name based tamper-detection mechanism in each exploit binary. Besides, although no packing and obfuscation techniques are employed, all strings are encrypted and there are no informative function names.

Security Weaknesses

- *Weak protection for the device-side app.* Unlike P1, even the original apk downloaded directly from P4's official website has little protection — only some basic class and function name obfuscation is used. The major body of SMALI code is still highly readable and has given out all detailed functional and encryption/decryption logic involved in *STEP 1* and *STEP 2*. For instance, reference strings such as "md5" are not encrypted, which dramatically accelerates the reverse engineering progress.
- *Debug output turned on in ELF binaries.* The exploit binaries will output decoded strings (*e.g.,* path of a vulnerable device driver) directly to the console. Obviously the developers forgot to turn off the debug option carelessly and this mistake significantly eases the task of locating the string decode and tamper-detection procedures. The protection in *STEP 3* is thus greatly weakened. Unfortunately, besides P4, we also find informative debug output in other providers' exploit binaries.

4.5.3 Small Root Providers

Small providers typically have only the device-side Android app, besides, they usually contain few but highly specialized exploits. In our survey, P6 and P7 are classified as small providers, both of their Android apps simply invokes the native exploit binary and the entire procedure has no protection. For the exploit binaries, although there are certain protections such as code obfuscation and tamper detection, they are generally primitive and easy to spot and bypass. Interestingly, the lack of effective defenses also makes it possible for larger root providers to take small providers' exploits and integrate them directly into their own products, as we observed in two larger providers.

4.6 Characterization and Case Studies of Exploits

As shown in Table 4.3, several top providers offer a large selection of root exploits. In this section, we dissect the 167 unique exploits from the largest provider P1 by beginning with the methodology to collect the exploits.

Exploits Collection Methodology. To download exploits from P1's online database, we need to provide sufficient information of different device models to P1's remote server (See Figure 4.5). Without access to a large number of real Android devices, we resort to online sources and factory images publicly available [135]. After crawling several such websites, we collect 5742 sets of device information and 2458 unique phone models with kernel ranging from 2.6.32.9 to 3.10.30 and Android version from 2.3.4 to 5.0.2. The list covers all mainstream manufacturers such as SAMSUNG, HTC, and SONY. They allow us to download 167 unique exploit binaries. Note that large providers claim to support over 10,000 or 20,000 Android device models and therefore the number of exploits we obtained may be far smaller than the actual number. Nevertheless, 167 unique exploit samples are still impressive from only 2458 phone models.

4.6.1 Breakdown of Exploits

Families of exploits. We hypothesize that these binaries are of high value to attackers since the number appears to be much larger than what we can find publicly. However, there is an important caveat that multiple binary files may simply be variations and adaptations of the same core exploit. In order to perform a fair comparison, we need a way to group similar binaries into families. Fortunately, the decrypted descriptor files returned from P1's server, as mentioned in Section §4.5.1, have an internal naming scheme to identify each exploit. An example of the internal name is *exploit98-3.2-v1*, in which “98” is used to number the exploit type, “3.2” is a specific kernel version, and “v1” indicates that this exploit is the 1st variation of an original exploit. Based on the naming scheme, we estimate that 59 different families exist, which is more than the 37 abusable public exploits still (See §4.2.3). From the naming schemes, we can also estimate the current size of P1’s exploit families to be in the hundreds.

Based on the knowledge gained from the public exploits targeting different layers, we analyze P1’s exploit binaries and its logic, *e.g.,* system calls and their parameters, we can classify a large portion of them into two main categories: 20 families belonging to kernel layer, typically featured with the use of vulnerable system calls such as `futex` (CVE-2014-3153) and `perf_event_open` (CVE-2013-2094), and 37 families belonging to driver layer, featured with operations on vulnerable device files such as /dev/exynos_mem. The remaining 2 families are difficult to classify. In the kernel layer, we have identified 17 families that can be mapped to publicly available exploits, but are unable to fully analyze the other 3. According to the exploit descriptors, we cannot locate the exploit for most affected devices from any public sources. For the driver layer exploits, we recognize 22 families as already published, but surprisingly, as we will discuss later, the remaining 15 families are potentially new exploits. Interestingly, we did not encounter any application layer exploits that typically check the existence of a process by name or a well-known file path.

New Driver-Layer Exploits. We identify 37 families of driver exploits, All driver layer exploits have the standard behavior of *open()* on a vulnerable device file in the form of

Kernel Version	2.6.32.9	2.6.35.7	3.0.15	3.0.16	3.0.31	3.0.8	3.4.0	3.4.39	3.4.43	3.4.5	3.4.67	3.10.0	3.10.9	3.10.30
Variants Count	2	2	1	1	3	10	21	8	1	9	2	1	1	3

* The calculation is based on our own collection, actual amount of P1's variants may be larger.

Table 4.4: P1's adaptation for CVE-2014-3153 (based on kernel version)

/dev/file followed by *ioctl()* or other syscalls on the file. We differentiate the device file name as kernel's built-in device or vendor-specific device and include the latter only. Even though many of them match existing exploits, we do find 15 new exploit families targeting 10 vulnerable unique device names. We are able to locate the unique device file name to specific devices which match the affected device models in P1's own exploit file descriptor. The affected devices include popular brands like SAMSUNG and some new models released less than a year ago. Due to security reasons we do not reveal the vulnerable device file names. Interestingly, in a recent research [135], it is suggested that vendor customizations of Android introduce considerable driver-layer vulnerabilities (no root exploit is discovered however). In our study, we find that such vulnerabilities can in fact lead to root privilege escalation. In retrospect, now that kernel exploits are harder and harder to come by with the latest OS security technology in place, it is natural to target the drivers to develop new exploits.

Adaptation. The most noticeable exploit family in P1's database is the one with 89 variants. By reverse engineering the exploit files we identified the family as implementations of CVE-2014-3153, the well-known "futex" kernel vulnerability. This confirms the need of adaptation of exploits as discussed in §4.3. To understand why this many variants are developed, we analyze the intended kernel version targeted by the 89 exploits and find 14 different kernel versions are targeted. Even for the same kernel version, P1 will apply different variants according to the kernel "build information" (*e.g.,* [#1 SMP PREEMPT Wed May 15 23:25:44 KST 2013]). The result is summarized in Table 4.4, P1 has covered most major linux kernel versions used by Android. For some popular versions such as "3.4.0", there exist more variants. Beside the adaptation for kernel versions, from the exploit descriptors, we also see that some variants specifically designed for certain device manufactures such as SAMSUNG and HUAWEI. The rest of the exploit families do not have many variants, *e.g.,* 42 families have only one binary.

Overall, we are impressed by the scale at which exploits are engineered by P1. It will be extremely difficult for an individual to match the amount of resources and engineering effort. We believe similar high impact kernel-level vulnerability such as CVE-2015-3636 (currently pioneered as PingPong Root) may be of similar value and require substantial adaptation effort. It is reported that the root exploit affects many latest Android devices such as Samsung Galaxy S6 and HTC One (M9) and the list is growing.

Timeline to add new exploits. As we have shown, exploits from large commercial root providers largely overlap with publicly available ones. One important metric indicating the competitiveness of the providers is the time it took from the date original exploits were first published to the date that they are incorporated. Even though there is no comprehensive data, we do have a unique data point on the latest PingPong root exploit, which was first published in May 2015. The same exploit is incorporated in P1 roughly two to three months later. We note that PingPong root is techinically intrisnic and involved. It is impressive that the provider has finished reverse engineering, developing, and testing of the exploit within such a short period. In fact, the incorporation happened before the full technical detail of the exploit is released and any proof-of-concept code is available. This demonstrates that commercial root providers are capable and swift, which is another reason why they may become targets of attackers.

4.6.2 Interaction with Advanced Security Mechanism

In 109 out of 167 P1's exploits (including all "futex" exploits), we observe special treatment for SELinux, which forms the base of the advanced Android security mechanisms such as SEAndroid [115] and Knox [31]. To support fine-grained mandatory access control, SELinux introduced the concept of "security context" whereby a process running as root may still be subject to restrictions imposed by the policies on the "context" it is running as. This effectively eliminates the powerful root in the traditional Linux. However, in AOSP, SELinux policy in Android 4.4 and below generally make the app domain either permissive or unconfined. Unless customized by vendors such as Samsung, it means that SELinux is effectively "disabled". Furthermore, even when SELinux policies are configured to be enforcing, as is done since Android 5.0 (AOSP), kernel-layer exploits can subvert SELinux easily by overwriting the related kernel data structures, given SELinux operates under the assumption that the kernel is intact. Specifically, almost in all cases of the 109 exploits, they overwrite not only the uid but also the sid and osid so that the security context effectively becomes "init", which is the most privileged one and is able to access almost all system resources. After that, they will write to $/proc/self/attr/current$ to change its string representation of security context to "u:r:init:s0".

Similarly, we also observe modifications to Linux "process capability" related k-ernel data structures such as $cap\_effective$ to 0xFFFFFFFF. Since process capabilities in

Linux share the same threat model as SELinux, it is not hard to imagine that they can be subverted in the same way.

When compared with public proof-of-concept exploit source code, we rarely find such level of care in dealing with the additional restrictions set by SELinux and process capabilities.

4.6.3 Anti-Virus Test

Since the root exploits are highly sensitive and may be leveraged by various Android malware, it is expected that anti-virus software on Android platform can identify most of them, including the ones implemented by root providers. We select the same 4 representative Android anti-virus products to test P1's 167 exploits. Because originally downloaded exploits from P1's database have packed the actual exploit code and employed a tamper-detection mechanism, we crafted 3 different versions for every exploit: 1) Original exploit fetched directly from P1's servers, with packing and tamper-detection on. 2) Unpacked exploit, which will expose all actual exploit logic to anti-virus products. 3) Repacked exploit with tamper-detection disabled. The last version can be highly dangerous since it can be utilized freely by malware that can unpack and execute at run time.

To test the antivirus software, we embed all exploit files of one version at a time in a self-developed Android app to trigger the proper scan. We test one antivirus software at a time. If a message is prompted that the app is safe to open, it indicates the antivirus software fails to detect any exploit in the app, and we uninstall the antivirus and install

	AVG	Lookout	Norton	Trend Micro
Original	N	N	N	N
Unpacked	N	N	N	13
Bypassed	N	N	N	N

* N:No threat detected

† All anti-virus at newest version when testing.

Table 4.5: Anti-Virus Test Results on P1's exploits

the next. If an alert is given flagging our app as malicious, we attempt to identify which subset of exploits are flagged by embedding one exploit file at a time and retest.

The test results are shown in Table 4.5. It is disappointing to see that no packed exploit is detected by any antivirus software. It is likely due to the custom obfuscation implemented by the provider that is not recognized. However, even for the unpacked ones, only Trend Micro can recognize 13 out of 167 exploit files as malicious. It is worth mentioning that the highly dangerous futex exploits as well as the PingPong root exploit are not caught by any antivirus software. This contrasts with the result in §4.2.4 where the public PingPong root exploit is in fact detected by Trend Micro. This suggests 1) P1's implementation of PingPong root is sufficiently different; 2) Trend Micro uses some kind of signature-based detection that is not effective at catching variants of the same exploit. Overall, the result shows that the state-of-the-art security products on Android platform still cannot address root exploits effectively. Worse, packing and obfuscation can easily evade detection.

4.7 Related Work

Android malware analysis. Android malware detection and analysis attracted much attention of research in the past few years [136, 95, 137, 44]. The Android Malware Genome

Project [136] has offered a public dataset of Android malware from year 2010 to 2012. The analysis covers behaviors from privacy invasion, financial charges, remote control, to root exploits embedded in the malware. In the ANDRUBIS [95] project, a more recent malware set of 400K samples collected between 2012 and 2014 is examined. It gives a more up-to-date view, yet little discussion is on malware carrying root exploits. Other sources include Contagio minidump [9] and VirusTotal [38]. DroidRanger [137] and DREBIN [44] attempt to detect Android malware by leveraging a carefully selected set of features.

So far, no evidence has shown that a single piece of malware embeds a large number of root exploits, likely because of the engineering challenge, *e.g.,* many exploits need to be adapted to work on specific device models. In our study, it is alarming to see the potential that malware can abuse the root provider logic to achieve this goal.

Android root exploits and defense. While a comprehensive characterization of Android root exploits is lacking, point studies have shown that root exploits were indeed abused by malware in the wild [137, 136]. As described in the Android Malware Genome Project [136], 36.7% of 1260 malware samples had embedded at least one root exploit.

Root providers present a unique position in computer history that they legitimately collect and distribute a large number of fresh root exploits. In theory, all commercial root providers should provide adequate protections on the exploits. In practice, unfortunately, as long as one of the providers fails to achieve that, malware authors can successfully "steal" the well engineered, adapted, and tested exploits against a diverse set of Android devices.

The development of Android comes with much improvement in Security. SeAndroid was shown to be effective against many root exploits that target user-level applications [115]. Research proposals use dynamic behaviors such as system calls and other events to detect root exploits [137, 77, 104]. Unfortunately, none of them is bullet-proof. For instance, new exploits such as TowelRoot and PingPong Root are not impacted by SeAndroid since they exploit kernel-layer vulnerabilities directly. In addition, the more expensive dynamic analysis techniques require root privilege to operate which limits their applicability and not to mention its impact on battery life.

Reverse engineering and anti-reverse engineering. Anti-reverse engineering aims at transforming a program into a semantically equivalent one yet much more difficult to comprehend and reverse engineer [59, 58]. Encryption, packing, symbol stripping, instruction reordering, *etc.* are commonly used obfuscation techniques against reverse engineering [58]. The key used to encrypt the code can either be embedded directly in the binary or burned onto the hardware [46, 119]. Most programs simply embed the key directly in its binary, including the famous STUXNET [66] and the binaries in root providers, as there is little support from the hardware to encrypt and decrypt instructions on the fly in general-purpose computing systems [119]. In addition, a different program binary encrypted with a different key needs to be distributed for every machine, which can be costly and complex to manage. More advanced techniques such as VM-based software protection [46] also exist. They dramatically increase the cost of reverse engineering by employing custom instruction set architectures.

In response to such anti-reverse engineering and obfuscation techniques, deobfuscation techniques are also developed [90, 108, 46]. It is not clear when such arms race will end as fundamentally the code under protection has to run physically on a machine controlled by the adversary. In a recent work, Zeng *et al.* proposes to use trace-oriented programming to implement binary code reuse [129]. The idea is that as long as the execution trace is observed and recorded at runtime, they can be extracted and reused. In principle, such ideas can be applied to extract obfuscated root exploit code. However, it is not sufficient as the self-verification logic still needs to be identified and removed.

On Android, the situation is not much different except some advanced obfuscation tools such as custom-VM-based protections are not yet available. As most large root providers need to protect both PC-side software and device-side app, the obfuscation strength is determined by the weaker side.

4.8 Discussion and Conclusion

Ethics. The study on root providers can be controversial. We study them because of two reasons: 1) root provider is a unique product in history that has unique characteristics. 2) Although legitimate, the functionality is implemented by exploiting vulnerabilities of the target system, which presents significant security risks. The goal of our research is to understand and characterize the risk that well-engineered exploits from the root providers can be stolen and easily repackaged in malware. By studying the protection mechanisms employed by root providers, we aim to quantify their strength and point out areas of weaknesses.

To protect the root providers, in the chapter, we intentionally anonymize their names when detailed results are shown. Further, we plan to release our findings to the corresponding root providers and device vendors.

Android-side root vs. PC-side root. In this study, we cover in detail mostly the root exploits implemented directly on Android devices. Most large root providers in fact offer both PC-side as well as Android-side root methods. The reason we focus on Android-side root is its risk of being abused by malware. It is worth mentioning that, as demonstrated by recent studies [121], a compromised PC can infect the mobile devices connected to it. Under such threats, the PC-side root exploits also become dangerous and are subject to abuse by PC malware. We leave this for future study.

Conclusion. In this chapter, for the first time, we uncover the mysterious Android root providers. We find they not only make significant efforts to incorporate and adapt existing exploits to cover more devices, but also craft new ones to stay competitive. However, these well-engineered exploits are not well protected, it is extremely dangerous if they fall in the wrong hands. This may also trigger a public policy/legal discussion on whether to regulate such companies that manufacture up-to-date exploits that are freely distributed.

Chapter 5

Improve Kernel Maintenance: Precise and Accurate Patch Presence Test for Binaries

5.1 Introduction

The number of newly found security vulnerabilities has been increasing rapidly in recent years [13], posing severe threats to various software and end users. The main approach used to combat vulnerabilities is patching; however, it is challenging to ensure that a security patch gets propagated to a large number of affected software distributions, in a timely manner, especially for large projects that have multiple concurrent development branches (*i.e.,* upstream versus downstream). This is due to the heavy code reuse in modern software engineering practice [83, 94, 89]. Thus, the capability to test whether a certain security patch is applied to a software distribution is crucial, for both defenders and attackers.

To better facilitate the discussion of this chapter, we differentiate the goal and scope of *patch presence test* from those of the more general *bug search*. Patch presence test, as its name suggests, checks whether a specific patch has been applied to an unknown target, assuming the knowledge of the affected function(s) and the patch itself, *e.g.,* "whether the heartbleed vulnerability of an openssl library has been patched in the `tls1_process_heartbeat()` function". Bug search, on the other hand, does not make assumptions on which of the target functions are affected and simply look for all functions or code snippets that are similar to the vulnerable one, *e.g.,* "which of the functions in a software distribution looks like a vulnerable version of `tls1_process_heartbeat()`." Our study focuses on the more specific problem of *patch presence test*, which aims to offer a precise and accurate answer. With this in mind, both lines of work have been studied in the following contexts:

Source to source. This type of work operates purely on source code level. Source code is required for both the reference and target. In recent studies, it is also typically assumed that patches about specific bugs are available.

Binary to binary. These work do not need any source code. Both the reference and target are in binary, thus all comparisons are based on binary-level features only. It does not assume the availability of patch information (about which binary instructions are related to a patch).

In this chapter, we consider a new category of **"source to binary"**, which is a middle ground between the above two, based on the following observations. First, open source has become a trend in computer world nowadays with an exploding number of software open sourced with full history of commits and patches (*e.g.,* hosted on github) [17]. In fact, most of the binary-only bug search studies include software such as Linux and openssl. Second, many open-source code or components are widely reused in closed-source software, *e.g.,* libraries and Linux-based kernels in IoT firmware [68, 105]. This is a critical change that allows us to leverage the source-level insight that can inform the binary patch presence test.

Unfortunately, the closely related work on binary-only bug search misses an important link in order to be twisted to perform accurate patch presence test. Due to its extremely large scope, they are forced to use similarity-based fuzzy matching (inherently inaccurate) to speed up the search process, instead of the more expensive yet more accurate approaches. As a result, most of the existing solutions usually take the whole functions for comparison [105, 106, 68, 127]. However, since security patches are mostly small and subtle changes [118], similarity-based approaches cannot effectively distinguish patched and un-patched versions.

In this chapter, we propose FIBER, a complementary system that completes the missing link and takes the similarity-based bug search to the next level where we can perform precise and accurate patch presence test. Fundamentally, FIBER addresses the following technical problem: "how do we generate binary signatures that well represent the source-level patch"? We address this problem in two steps: First, inspired by typical human analyst's behaviors, we will pick and choose the most suitable parts of a patch as candidates for binary signature generation. Second, we generate the binary signatures that preserve as much source-level information as possible, including the patch and the corresponding function as a whole.

We summarize our contributions as follows:

(1) We formulate the problem of patch presence test under "source to binary", bridging the gap from the general bug search to precise and accurate patch presence test. We then describe FIBER — an automatic, precise, and accurate system overcoming challenges such as information loss in the binaries. FIBER is open sourced[*].

(2) We design FIBER inspired by human behaviors, which picks and chooses the most suitable parts of a patch to generate binary signatures representative of the source-level patch. Besides, the test results can also be easily reasoned about by humans.

(3) We systematically evaluate FIBER with 107 real word vulnerabilities and security patches on a diverse set of Android kernel images [†] with different timestamps, versions and vendors, the results show that FIBER can achieve high accuracy in security patch p-resence test. We discover real-world cases where critical security patches fail to propagate to the downstreams.

5.2 Related Work

In this section, we discuss the related work primarily under bug search and how they are currently applied to the patch presence test problem. We divide them as source-level and binary-level.

[*]https://fiberx.github.io/

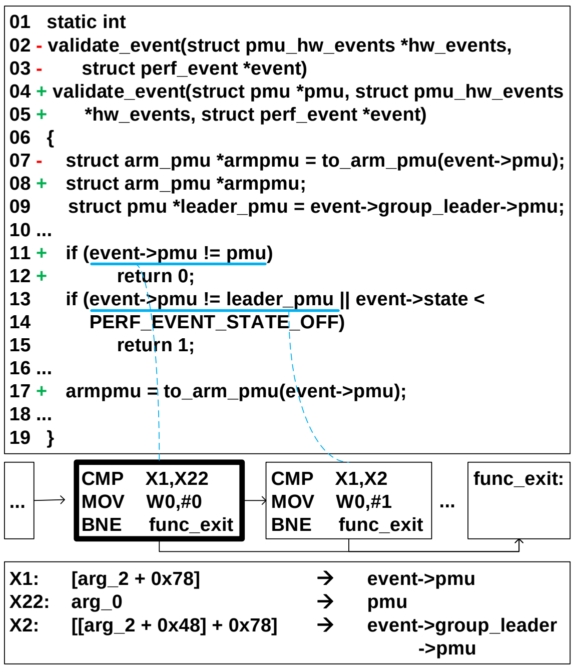
[†]Although Android follows open-source license, many Android device vendors still do not publish their source code or only do that periodically (with significant delays) for certain major releases.

Source-level bug search. Many studies focused on finding code clones both inside a single software distribution and across distributions [87, 93, 84, 83, 89]. The general goal is to find code snippets similar to a given buggy one — a more general goal that can be twisted to also conduct patch presence test. Since bug search typically does not limit the search scope to only a single function, it needs to face potentially millions of lines of code in large software [83]. Due to the scalability concern, bug search solutions are typically framed as some form of similarity matching using features extracted from the source code, including plain string [49], tokens [87, 93, 83, 89], and parse trees [84]. Unfortunately, this makes it challenging to ascertain whether the identified similar code snippets have been patched; this is because the patched and un-patched versions can be similar (especially for security patches that are often small) [83].

Binary-level bug search. Similar to the source-level work, binary-level approaches follow a similar principle of finding similar code snippets. To overcome the challenge of lack of source-level information, *e.g.,* variable type and name, these solutions need to look for alternative features such as structure of the code [88, 68, 127]. Since the "binary to binary" bug search does not assume the availability of symbol tables, they are forced to check out every single function in the target even if it only intends to conduct an accurate patch presence test on a specific function. For example, given a vulnerable function, Genius [68] and Gemini [127] are essentially looking for the same affected function(s) in the complete collection of functions in a target binary. Due to the scalability concern again, these features and solutions are engineered for speed instead of accuracy. BinDiff [8] and BinSlayer [50] check the control flow graph similarity based on isomorphism. As more advanced solutions, Genius [68] and Gemini [127] extract feature representations from the control flow graphs and encodes them into graph embeddings (high dimensional numerical vectors), which can speed up the matching process significantly. Unfortunately, under the huge search space, more accurate semantics-based solutions are not believed to be scalable [68, 127]. For instance, Pewny *et al.* [105] computes I/O pairs of basic blocks to match similar basic blocks in a target function. BinHunt [70] and iBinHunt [100] use symbolic execution and theorem provers to formally verify basic block level semantic equivalence.

FIBER is in a unique position that leverages the source-level information to answer a more specific question — whether the specific affected function is patched in the target binary. To our knowledge, Pewny *et al.*'s work [105] is the only one that claims source-level patch information can be leveraged to generate more fine-grained signatures for bug search (although no implementation and evaluation). However, its goal is still focused on bug search instead of patch presence test, which means that it still attempts to search for similar (un-)patched code snippets (in binary) in the entire target, making it too fuzzy to answer the problem of patch presence test.

Finally, binary-level bug search has been extended to be cross-architecture [106, 105, 68, 127]. FIBER naturally supports different architectures with the assumption that source code is available, allowing us to generate different signatures for different compiled binaries.



```
   static int
 - validate_event(struct pmu_hw_events *hw_events,
 -       struct perf_event *event)
 + validate_event(struct pmu *pmu, struct pmu_hw_events
 +       *hw_events, struct perf_event *event)
   {
 -   struct arm_pmu *armpmu = to_arm_pmu(event->pmu);
 +   struct arm_pmu *armpmu;
     struct pmu *leader_pmu = event->group_leader->pmu;
 ...
 +   if (event->pmu != pmu)
 +           return 0;
     if (event->pmu != leader_pmu || event->state <
         PERF_EVENT_STATE_OFF)
             return 1;
 ...
 +   armpmu = to_arm_pmu(event->pmu);
 ...
   }
```

```
...  →  CMP   X1,X22    →  CMP   X1,X2     ...  func_exit:
        MOV   W0,#0        MOV   W0,#1
        BNE   func_exit    BNE   func_exit
```

```
X1:    [arg_2 + 0x78]             →    event->pmu
X22:   arg_0                      →    pmu
X2:    [[arg_2 + 0x48] + 0x78]    →    event->group_leader
                                           ->pmu
```

Figure 5.1: Patch of CVE-2015-8955

5.3 Overview

In this section, we first walk through a motivating example to summarize FIBER's general intuition, then position FIBER in a larger picture.

A motivating example. We pick the security patch for `CVE-2015-8955`, a Linux kernel vulnerability, to intuitively demonstrate a typical workflow of patch presence test which FIBER closely emulates. The patch is shown in Fig 5.1.[‡] To test whether this patch exists in the target binary, naturally we will follow the steps below:

Step 1: Pick a change site (*i.e.*, sequence of changed statements). At first glance, we can see that the patch introduces multiple change sites. However, not all of them are ideal for

[‡]For simplicity, we include only one of the two changed functions in the patch and removed comments and context lines. The full patch can be found in [32].

the patch presence test purpose. Line 1-5 adds a new parameter "`pmu`" for original function, which will be used by the added "if" statement at line 11. Another change is to move the assignment of "`armpmu`" from line 7 to line 17. The "`to_arm_pmu()`" used by the assignment is a small utility macro, which will result in few instructions without changing the control flow graph (CFG), making it difficult to be located at binary level. However, the added "if" statement at line 11 will introduce a structural change to the CFG, besides, it also has a unique semantic as it involves the newly added function parameter. Therefore, it is natural to consider line 11 a more suitable indicator of patch presence.

Step 2: Rough matching. Now we have decided to search in the target binary function for the existence of line 11 in Fig 5.1, typically we will start from matching the CFG structure since it is easy and fast. This step can be similarly carried out in the source code level also. Specifically, one condition in the "if" statement will generally lead to a basic block with two successors, Thus for line 11, we will first try to locate those basic blocks with out-degrees of 2. Besides, one successor of the basic block should be the function epilogue since at line 12 the function will return if passing the checks at line 11. In Fig 5.1 we also show a part of the CFG generated from a patched Android kernel image, we can see that both the bolded basic block and the basic block right of it satisfy this requirement.

Step 3: Precise matching. Out of the two candidate basic blocks in the target binary, we now should need some semantic information to further distinguish them. Ideally, if we have the source level information such as variable names, a human can typically make a decision already (assuming the target function does not change variable names). With limited information at the binary level, we need to map the binary instructions to source-level statements somehow. This is usually a time-consuming process for human analysts, since they typically need to understand which register or memory location corresponds to which source-level variable. Following the same example in Fig 5.1[§], an analyst needs to inspect the registers used in the "`cmp`" instruction of candidate blocks. Specifically, by tracking the register's origin (listed at the bottom of Fig 5.1), we can finally tell the differences of the two "`cmp`" instructions and correctly decide that the bolded basic block is the one that maps back to line 11.

System architecture. Fig 5.2 illustrates the system architecture, which is abstracted from human analysts' procedure. It has four primary inputs: (1) the source-level patch information; (2) the complete source code of a reference; (3) the affected function(s) in the compiled reference binary; (4) the affected functions in the target binary. It is obvious that (4) is readily available if the symbol table is included in the target binary (*e.g.,* true in most Linux-based kernel images). However, in the more general case we do not make this assumption, neither do the state-of-the-art binary-only bug search work [68, 127, 105]. Fortunately, these similarity-based approaches solve this very problem by identifying functions in the target binary that look similar to a reference one, thus the symbol table of the target binary can actually be inferred — in addition to research studies [68, 127], BinDiff [8] also has a built-in functionality serving this purpose. We leave the integration of such functionality into FIBER as future work, since all kernel images as test subjects in our evaluation have embedded symbol tables.

[§] We use AArch64 assembly instructions in this example, if not explicitly stated, the same assembly instructions will also be used in all other examples across the chapter.

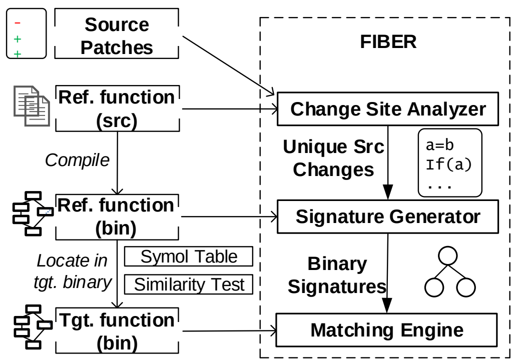


Figure 5.2: Workflow of FIBER

This shows that the similarity-based bug search and the more precise patch presence test are in fact not competing solutions; rather, they complement each other. The former is fast/scalable but less accurate; the latter is slower but more accurate. In a way, bug search acts as a coarse-grained filter and outputs a ranked list of candidate functions which can be used as input (4) of FIBER for further processing. Since the search space of FIBER is now constrained to only a few candidate functions (one if with symbol table), it opens up the more expensive analysis.

With the inputs in mind, we now describe the three major components in FIBER: (1) Change site analyzer. A single patch may introduce more than one change site in different functions and one change site can also span over multiple lines in source code. Change site analyzer intends to pick out those most representative, unique and easy-to-match source changes by carefully analyzing each change site and the corresponding reference function(s), mimicking what a real analyst would do. Besides, during this process, we can also obtain useful source-level insight regarding the change sites (*e.g.,* the types of statements and the variables involved), which can guide the later signature generation and matching process.

(2) Signature generator. This component is responsible for translating source-level change sites into binary-level signatures. Essentially this step requires an analysis to ensure that we can map binary instructions to source-level statements, which is challenging because of the information loss during the compilation process. The key building block we leverage is binary symbolic execution for this purpose.

(3) Matching engine. The matching engine's task is to search a given signature in the target binary. To do that, we first need to locate the affected function(s) in the target binary with the help of the symbol table. Then the search is done by first matching the syntax represented by the topology of a localized CFG related to the patch (a much quicker process), and then the semantic formulas (slower because of the symbolic execution). This process is similar to the one described in the motivating example.

It is worth noting that as long as a signature is generated for a particular security patch, it can then be saved and reused for multiple target binaries, thus we only need to run the analyzer and generator once for each patch.

Scope. (1) FIBER naturally supports analyzing binaries of different architecture and compiled with different compiler options. This is because of the availability of source code, which allows us to compile the source code into any supported architecture with any compiler options. More details will be discussed in §5.5 and §5.6.

(2) FIBER is inherently not tied to any source language although currently it works on C code. We do require debug information to be generated (for our reference binary) by compilers that can map the binary instructions back to source level statements as will be discussed in §5.4.3. All modern C compilers can do this for example.

Potential users and usage scenarios. We envision third-party auditors/developers will be FIBER's primary users, such as independent security researchers, security companies, software integration companies that rely on code/binaries supplied from others. Even for first-party developers, checking security patches at the binary level offers an extra layer of safety. As will be shown in §5.6.4, some vendors indeed forgot to patch critical vulnerabilities even though they have source access (*i.e.,* human errors), while systems like FIBER could have caught it.

5.4 System Design

In this section, we describe FIBER's design in depth by walking through the requirement of signatures and the design of each component.

5.4.1 Signature

The signature is what represents a patch. In general, we have two criterion for an "ideal" signature:

(1) Unique. The signature should not be found in places other than the patch itself. Otherwise, it is not unique to the patch. Specifically, it should not exist in both the patched and un-patched versions. This means that the signature should not be overly simple, which may cause it to appear in places unrelated to the patch.

(2) Stable. The signature should be robust to benign evolution of the code base, *e.g.,* the target function may look different than as the reference due to version differences. This means that the signature should not be overly complex (related to too many source

lines), which is more likely to encounter benign changes in the target, creating false matches of the signature.

As we can see, the above two seemingly conflicting requirements ask for a delicate balance in signature generation, which we will elaborate in this section. Fundamentally, we need to pick a unique source change from a patch for which we believe a corresponding binary signature can be generated that well represents it. What works in our favor is that the reference and target function should share significant variable-level semantics. Assuming both versions are patched, things like "how a variable is derived and dereferenced" and "how a condition is derived" should be the very much the same. The binary signature simply need to carry this necessary information to recover the semantics present in the source.

Informally, we define a binary signature to be a group of instructions, that not only structurally correspond to the source-level signature, but also are annotated with sufficient information (*e.g.,* variable-level semantics) so that they can be unambiguously mapped to the original source-level change site. We will elaborate the translation process in §5.4.3.

5.4.2 Change Site Analyzer

The input of the change site analyzer is a source patch and the reference code base. It serves two purposes. (1) Since a patch may introduce multiple change sites within or across different functions, the analyzer aims to pick a suitable signature according to the criterion mentioned in §5.4.1. (2) Another goal is to gain insights of the patch change sites, from which the binary signature generator will benefit. We divide this process into two phases and detail them as below.

5.4.2.1 Phase 1: Unique Source Change Discovery

A patch can either add or delete some lines, thus we can either changes based on either the absence of patch (*i.e.,* existence of deleted lines) or presence of patch (*i.e.,* existence of added lines). For the purpose of discussion, we assume that our signature generation is based on the presence of patch and focused on the added lines; the opposite can be done similarly. The general strategy is to start from a single statement and gradually expand if necessary. For each added statement in the patch, the following steps will be performed:

(1) Uniqueness test. Basically, a statement has to exist in only the added lines of the patch and nowhere else (*e.g.,* un-patched code bases)". For this, we can apply a simple token-based sequence matching using a lexer [83]. We wish to point out that this uniqueness test captures not only token-based information but also semantic-related information. For instance, the example source signature in Fig 5.1 at line 11 encodes the fact that the first function parameter is compared against a field of the last parameter, and this semantic relationship is unique (which we need to preserve in binary signatures).

(2) (*optional*) Context addition. If no single statement is unique, we consider all its adjacent statements as potential context choices. The "adjacent" is bi-directional and on the control flow level (*e.g.,* the "if" statement has two successors and both of which can be considered the context), thus there can be multiple context statements. We gradually expand the context statements, *e.g.,* if one context statement is not enough, we try two.

(3) Fine-grained change detection. By convention, patches are distributed in the form of source line changes. Even when a line is partially modified, the corresponding patch

will still show one deleted and one added line. We detect such fine-grained changes within a single statement / source line, by comparing it with its neighbouring deleted/added lines. This is to ensure that we do not include unnecessary part of the statement which will bloat the signature. For example, if only one argument of a function call statement is changed, we can ignore all other arguments in the matching process to reduce potential noise, improving the "stability" of the signature.

(4) Type insight. The types of variables involved in source statements are also important since it will guide the later binary signature generation and matching. Theoretically, we can label the type of every variable in the reference binary (registers or memory locations in the binary) and make sure the types inferred in the target match (more details in §5.4.3.1). However, sometimes type match is not good enough to uniquely match a signature. A special case is a const string which is stored statically at a hardcoded memory address. If the only change in a patch is related to the content of the string, then both binary signature generation and matching should dereference the `char*` pointer and obtain the actual string content; otherwise, the signature will simply contain a const memory pointer whose value can vary across different binaries. Even if the pointer type matches as `char*` in the target, it is still inconclusive if it is a patched or un-patched version (we give some real examples in §5.6 as case studies).

After the above procedure, we now have some unique and small (thus more stable) source changes.

5.4.2.2 Phase 2: Source Change Selection

Previous step may generate multiple candidate unique source changes for a single patch. In practice, the presence of one of them may already indicate the patch presence. In addition, some source changes are more suitable for binary signature generation than others. In FIBER, we will first rank all candidate changes and pick the top N for further translation. The ranking is based on three factors (from least important to most):

(1) Distance to function entrance. Short distance between statements in the source-level signature and the function entrance will accelerate the signature generation process because of its design which we will detail in §5.4.3.

(2) Function size. If the source code signature is located in a smaller function, the matching engine will benefit since the search space will be reduced and it is less likely to encounter "noise". In addition, the matching speed will be faster. Note that this is more important than (1) because the signature generation process is only a one-time effort while matching may be repeated for different target binaries.

(3) Change type. The kinds of statements involved in a change matters. As shown previously in §5.3, if the change involves some structural/control-flow changes (*e.g.,* "if" statement), we can quickly narrow down the search range to structurally-similar candidates in the target binary, affecting the matching speed. More importantly, it can also affect the stability of the binary signature. Unlike statements such as a function call, which may get inlined depending on the compiler options, structural changes in general are much more robust.

We categorize the source changes into several general types: (1) function invocations (new function call or argument change to an existing call), (2) condition related (new conditional statement or condition change in an existing statement), (3) assignments (which may involve arithmetic operations). Actual source changes can have multiple types, *e.g.,* a function invocation can have an argument derived from an assignment or follow a conditional statement. Generally, we rank “new function call” (if FIBER determines that it is not inlined in the reference binary[¶]) the highest because one can simply decide the patch presence by the presence of the function invocation, which is straightforward with the symbol table. We also rank “condition” related signatures (*e.g.,* “if” statement) high because it introduces both structural changes and semantic changes. On the other hand, a simple assignment statement, including assignment derived from arithmetic operations (*e.g.,* `a=b+c;`), will not affect the structure in general, so it is less preferred. Besides, pure control flow transfer (*e.g.,* addition of a “goto”) is not preferred as well since we may need to include extra context statements that are unrelated to the change site, which is less stable. Note that there are certain source-level changes are simply not visible at the binary level (*e.g.,* source code comments) or difficult to locate (variable declaration).

5.4.3 Signature Generator

We first need to compile the reference source into the reference binary, from which the binary signatures will be generated according to the selected unique source change. As discussed in §5.4.2, we will still assume that the signature is based on the patched

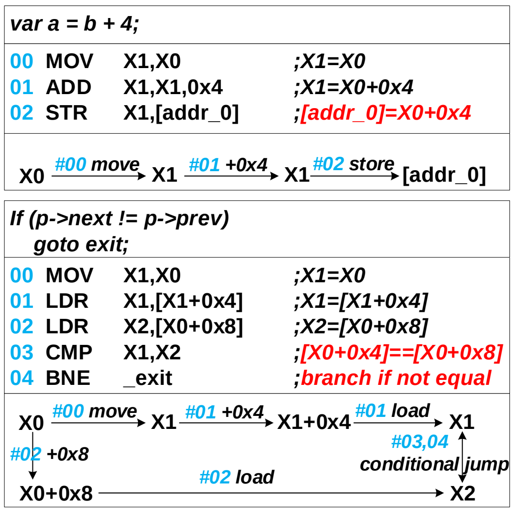
[¶] It looks the presence of the corresponding binary instruction that calls to the exact function.

version. Also, during the compilation process, we will retain all the compiler-generated debug information for future use.

5.4.3.1 Binary Signature Generation

Identify and organize instructions related to the source change. Given the reference binary, the first thing is to locate the corresponding binary instructions related to the source change. This can be done with the help of debug information since it provides a mapping from source code lines to binary instructions. We will then construct a local CFG that includes all the nodes containing the identified instructions, which is straightforward if these nodes are connected to each other, otherwise, we need to add some padding nodes to make a connected local CFG, which by nature is a steiner tree problem [82]. For this purpose we use the approximation steiner tree algorithm implemented in the NetworkX package [23]. The topology of such a local CFG reflects the structure of the original source change. Compared to full-function CFG, this local CFG structure is more robust to d-ifferent compiler options and architectures since it excludes unrelated code. That being said, compilation configurations may still affect the signature. Therefore, ideally we should use the same compilation configuration of the reference kernel as the target. As will be described in §5.6.1, we follow a procedure to actively probe the compilation configuration of the target kernel.

Identify root instructions. Theoretically all these instructions identified in the local CFG above will be part of the binary signature. However, this is not a good idea in practice as only a subset of instructions actually summarizes the key behavior (data flow semantic); we refer to such instructions as “root instructions”. The more instructions we include in a



```
var a = b + 4;

00  MOV    X1,X0            ;X1=X0
01  ADD    X1,X1,0x4        ;X1=X0+0x4
02  STR    X1,[addr_0]      ;[addr_0]=X0+0x4
```

$$\begin{tikzcd} X0 \arrow[r, "\#00\ move"] & X1 \arrow[r, "\#01\ +0x4"] & X1 \arrow[r, "\#02\ store"] & {[addr\_0]} \end{tikzcd}$$

```
If (p->next != p->prev)
   goto exit;

00  MOV    X1,X0            ;X1=X0
01  LDR    X1,[X1+0x4]      ;X1=[X1+0x4]
02  LDR    X2,[X0+0x8]      ;X2=[X0+0x8]
03  CMP    X1,X2            ;[X0+0x4]==[X0+0x8]
04  BNE    _exit            ;branch if not equal
```

$$\begin{tikzcd} X0 \arrow[r, "\#00\ move"] \arrow[d, "\#02\ +0x8"'] & X1 \arrow[r, "\#01\ +0x4"] & X1+0x4 \arrow[r, "\#01\ load"] & X1 \arrow[d, leftrightarrow, "{\#03,04\ conditional\ jump}"] \\ X0+0x8 \arrow[rrr, "\#02\ load"] & & & X2 \end{tikzcd}$$

Figure 5.3: Data flow analysis of example basic blocks

binary signature, the more specific and less "stable" it becomes. For instance, a compiler may insert additional "intermediate" instructions to free up some registers (by saving their values to memory). If we unnecessarily include all these instructions, we may not get a match in the target. Take the two source-level statements in Fig 5.3 as examples, the first statement is an assignment where 3 binary instructions are generated to perform the operation. However, capturing the last instruction alone is already sufficient, because we know through data flow analysis that `X1` is equal to `X0+0x4` and can therefore discard the first and second instruction. Similarly, instruction 03 and 04 corresponding to the second statement already sufficiently capture its semantic, because the outputs of instruction 00, 01 and 02 will later be consumed by other instructions.

Simply put, we define "root instructions" to be the last instructions in the data flow chains (where no other instructions will propagate any data further), along with some

Signature Type	Root Instructions (x86 example)
Function call	call,push
Conditional statement	cmp, conditional jmp
assignment (incl. arithmetic ops)	mov,add, sub,mul,bit ops...
Unconditional control transfer	jmp,ret

Table 5.1: Types of root instructions

complementary instructions that complete the source-level semantic. For instance, by this definition, the `cmp` instruction will be the root instruction. However, we need to complement it with the next conditional jump instruction to complete its conditional statement semantic. For function call instructions, the root instructions will include the push (assuming x86) of arguments (as they each become the last instruction in a data flow chain to prepare a specific argument), and the call instruction (to complete the function invocation semantic).

Note that compilers may still generate slightly different root instructions for the same statements (due to compiler optimizations, *etc.*). To facilitate signature matching, we deem root instructions equivalent as long as their types are the same (normalization of root instructions). We illustrate this in Table 5.1 where we show the different types of instructions that may be generated from the same source change. For instance, a compiler may choose to use bit operations instead of multiplications for an assignment statement.

Annotate root instructions. Now we need to make sure that the root instructions are sufficiently labeled (which is our binary signature) such that they can be uniquely mapped to source changes.

arg:	***function argument***
var:	***local variable***
ret:	***callee return value***
imm:	***immediate value***
[ ]:	***dereference***
op:	***binary operators***
expr:	***arg* | *var* | *ret* | *imm***
	| [*expr*] | *expr op expr*
	| if(*expr*) then *expr* else *expr*

Figure 5.4: Notation for formula (expression) annotating root instruction operands

Following the observation mentioned earlier in §5.4.1 that the target and reference function should share variable-level semantics (as they are simply different versions of the same function), we formulate the goal as *mapping the operands (registers or memory locations) of the root instructions back to source-level variables.* This is sufficient because if the target function indeed applied the patch, the variables related to the patch should be the same ones as what we saw in the reference function. Now, our only job here is to ensure that the binary signature retains all such semantic information. To this end, we compute a full-function semantic formula for each operand (up to the point of root instructions). As shown in Fig 5.1, these formulas are in the form of ASTs – essentially formulated as expressions following the notation in Fig 5.4.

Note that from a function's perspective, any operand in an instruction can really be derived from only four sources:

(1) a function parameter (external input), *e.g.,* `ebp+0x4` if it is x86, `X0` or `X1` if it is aarch64;

(2) a local variable (defined within the function), *e.g.,* `ebp-0x8` in x86 or `sp+0x4` in aarch64 (which use registers to pass arguments);

(3) return values from function calls (external source), *e.g.,* a register holding the return value of a function call;

(4) an immediate number (constant), *e.g.,* instruction/data address (including global variables), offset, other constants;

These sources all have meaningful semantics at the source level. The question is how do we leverage them in the binary signature. Do we require the binary signature to state something precise "the fourth parameter of the function is used in a comparison statement", or something more fuzzy "a local variable is dereferenced at an offset, whose result is passed to a function call"? These choices all have implications on the unique and stable requirement of the signature. We discuss how we handle these four basic cases:

(1) Function parameter. From the calling convention, we can at least infer where memory location corresponds to which parameter. Despite the fact that function prototpye may change in the target, our current policy assumes otherwise (as the change happens rather infrequently). As an extension, we could use the type of the parameter (as mentioned in §5.4.2), or even its usage profile to ensure the uniqueness of the parameter. Note that this would also require analysis of the target function to derive similar information (which will require more expensive binary-level type inference techniques [91, 61]).

(2) Local variable. This is similar to the function parameter case, except that local variables are much more prone to change, *e.g.,* new variables may be introduced. In theory, we could similarly use type information and the way the local variable is used to ensure the uniqueness the variable in the signature. For now, we do not conduct any additional analysis and simply treats all local variables as the same class without further

differentiation. Interestingly, we will show in §5.6 that this strategy already can generate signatures that are unique enough.

(3) Return values from function calls. This is a relatively straightforward case, we simply tag the return value to be originated from a specific function call.

(4) Immediate number. It is generally not safe to use the exact values of the immediate numbers, especially if it has to do with addresses. For instance, a `goto` instruction's target address may not be fixed in binaries. A field of a `struct` may be located at different offsets, *e.g.,* the target binary has a slightly different definition. We need to conduct additional binary-level analysis to infer if a target address is pointing to the right basic block (*e.g.,* by checking the similarity of the target basic block), or the offset is pointing to a specific field (*e.g.,* by type inference [91, 61]). Our current design allows for such extensions but at the moment simply treats immediate numbers as a class without differentiating their values, unless the values are related to source-level constants and unrelated to addresses, *e.g.,* `a = 0;`.

In our experience, we find that even without having a precise knowledge of these basic elements in the signature, the semantic formula that describe them is typically already unique enough to annotate the operands; ultimately allow us to uniquely map the root instructions to source-level statements. We show a concrete example in Fig 5.5 with both reference and target in comparison. As we can see, the patch line is in red: `a=n*m+2;`, a fairly straightforward assignment statement, which is used as a unique source change. In the binary form, we would identify the store instruction as root instruction, and annotate both operands accordingly. In this case, we know that `X3=X0*X1+0x2` which represents

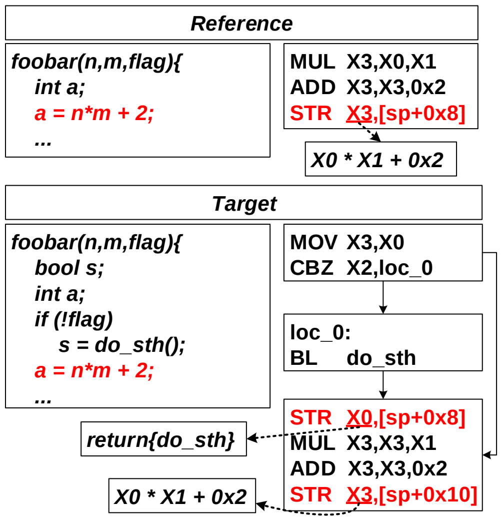


Figure 5.5: Illustration of the binary signature matching

`arg_0*arg_1+0x2` and it is being stored into a local variable at `sp+0x8`. Similarly, the target source has the same patch statement (and should be considered patched) even though it has also inserted some additional code with a new local variable. When we attempt to match the binary signature, there are three points worth noting:

First, the local variable `a` is now located at a different offset from `sp`, *i.e.,* `sp+0x10`. We therefore cannot blindly use a fixed offset to represent the same local variable across reference and target. Instead, we could apply the additional strategies mentioned above: (1) Inferring the type of local variables in the target binary and conclude that `sp+0x10` is the only integer variable and therefore must correspond to `sp+0x8`. (2) Profiling the behaviors of all local variables in the target binary and attempt to match the one most similar to `sp+0x8` in the reference. For example, we know `sp+0x8` in the binary (*i.e.,* `s`) takes the value from a function return, while `sp+0x10` (*i.e.,* `a`) did not (and `sp+0x10` is the more likely one). Interestingly, even if we do not perform the above analysis, the fact that there is a root instruction storing a unique formula `X0*X1+0x2` to a local variable (any) is already unique enough to be a signature that lead to a correct match in the target.

Second, to show that isolated basic block level analysis is not sufficient, we note the `mov` instruction in the first basic block of the target binary which saves X0 to X3 to free up X0 for the return value of `do_sth()`. It is imperative that we link X3 to X0 so that the final formula at the root instruction (*i.e.,* last instruction of the last basic block) will be the same as the one computed in the reference binary.

Third, there is an additional store instruction in the last basic block of the target binary, which saves X0 (return value of `do_sth()` to `sp+0x8` (*i.e.,* `s`). Note that this may look like a root instruction as well from data flow perspective. However, since it is attempting to store a return value instead of the formula in the original signature, it will not cause a false match.

5.4.3.2 Binary Signature Validation

Even though we have the best intention to preserve the uniqueness and stability of the selected source change, due to the information loss incurred in the translation, we still need to double check that the candidate binary-level signatures actually still satisfy the requirements.

(1) Unique. For each patch, we will prepare both the patched and un-patched binaries as references and then try to match the binary signature against them, with the matching engine (detailed in §5.4.4). For a binary signature based on the patched code, it will be regarded as unique only when it has no match in the reference un-patched binary.

A unique binary signature may still have multiple matches (although rare) in the reference patched binary, in this case, we will record the match count as auxiliary information. When using it to test the target binary in real world, only when the match count is no less than previously recorded one, will we say that the patch exists in target binary.

(2) Stable. Our previous effort in §5.4.2 to keep a small footprint of the unique source change can also help to improve the binary signature stability here, since the sizes of source change and binary signatures are related. Besides, we can also prepare multiple versions of patched and un-patched function binaries (if more ground truth data are available) and test the generated binary signature against them. This can help to pick out those most stable binary signatures that exist in all patched binaries but none of un-patched binaries.

5.4.4 Signature Matching Engine

Matching engine is responsible for searching a given binary signature in the target binary (*i.e.,* the test subject). This section will detail the searching process. As briefly mentioned in §5.3, we first need to locate the target function in the target binary by its symbol table, then we will start to search the binary signature in it. We divide the search into two phases: rough matching and precise matching.

Rough matching. This is a quick pass that intends to match the binary signature by some easy-to-collect features. These features include:

(1) CFG topology. The binary signature itself is basically a subgraph of the function CFG. This step is useful unless the binary signature resides in only a single basic block (*e.g.,* the signature for an assignment statement).

(2) Exit of basic blocks. In general each basic block has one of two exit types: unconditional jump and conditional jump, the former can be further classified into call, return, and other normal control flow transfer for most ISAs. Thus, basic blocks can be quickly compared by their exit types.

(3) Root instruction types. As described in §5.4.3.1, we will analyze each basic block in the signature and decide its root instruction set. The instruction types can then be used to quickly compare two basic blocks. This requires generating the data flow graph for each basic block in target function binary, which is more expensive than previous steps but still manageable.

With above features, we can quickly narrow down the search space in the target function. If no matches can be found in this step, we can already conclude that the signature does not exist, otherwise, we still need to precisely compare every candidate match further.

Precise matching. In this phase, we leverage the annotation produced in §5.4.3.1 to perform a precise match on two groups of root instructions. We essentially just need to compare their associated annotation (*i.e.,* semantic formulas).

To fulfill the semantic comparison, we first need to generate semantic formulas for all the matched candidate root instructions, which can be done in the same way as detailed in §5.4.3.1. If all formulas of the signature root instructions can also be found in the candidate root instructions, the two will be regarded as equivalent (*i.e.,* they map to the same source-level signature/statements).

To compare two formulas (essentially two ASTs), there have been prior solution-s that calculate a similarity score based on tree edit distance [67, 106]; however, FIBER intends to give a definitive answer about the match result, instead of a similarity score. Alternatively, theorem prover has been applied to prove the semantic equivalence of t-wo formulas [70], which definitely provides the best accuracy but unfortunately can be very expensive in practice. In this chapter, we choose a middle ground. Based on the observations that semantic formulas capture the dependency and therefore the order of in-structions cannot be swapped, we know that the structure of formulas is unlikely to change (our evaluation confirms this), *e.g.,* `(a+b)*2` will not become `a*2+b*2`. In addition, with normalization of the basic elements of the formula, the matching process is also robust to non-structural changes. Basically, the matching process simply recursively match the op-erations and operands in the AST, with some necessary relaxations (*e.g.,* if the operator is commutative, the order of the operands will not matter). We also simplify the AST with a Z3 solver [63] before comparison.

5.5 Implementation

We implement the prototype of FIBER with 5,097 LOC in Python on top of Angr [114], as it has a robust symbolic execution engine to generate semantic formulas. To suit our needs, we also changed the internals of Angr (including 1348 LOC addition and 89 LOC deletion). Below are some implementation details.

Architectural dependencies. As mentioned, FIBER in principle supports any architec-ture as we can compile the source code into binaries for any architecture. Further, since we use Angr which lifts the binaries into an intermediate language VEX (which abstracts away instruction set architecture differences), most of our system works flawlessly without the need of tailoring for architectural specifics. This not only allows FIBER to be (for the most part) architectural independent, but also facilitates the implementation. For instance, when searching for root instructions, the data flow analysis is performed on top of VEX. However, some small engineering efforts are still needed for multi-architectural support, such as to deal with different calling conventions. At current stage FIBER supports aarch64.

Root instruction annotation. To generate semantic formulas for root instruction operands, it is necessary to analyze all the binary code from the function entrance to the root instruction. We choose symbolic execution as our analyze method since it can cover all possible execution paths and obtain the value expression of any register and memory location at an arbitrary point along the path.

Symbolic execution is well known for the path explosion problem, which makes it expensive and not as practical. We employ multiple optimizations to address the performance issue as detailed below.

(1) Path pruning. Before starting the symbolic execution we will first perform a depth first search (DFS) in the function CFG to find all paths from the function entrance to the root instructions. We will then put only the basic blocks contained in these paths in the execution whitelist, all other basic blocks will be dropped by the symbolic execution engine. Besides, we also limit the loop unrolling times to 2 to further reduce the number of paths.

(2) Under-constrained symbolic execution. As proposed previously [107], under-constrained symbolic execution can process an individual function without worrying about its calling contexts, effectively confining the path space within the single function. Although

the input to the function (*e.g.,* parameters) is un-constrained at the beginning, it will not affect the extraction of the semantic formulas since they do not need such initial constraints. Un-constrained inputs may also lead the execution engine to include infeasible paths in real world execution, however, our goal for semantic formulas is to make them comparable between reference and target binaries, as long as we use the same procedure for both sides, the extracted formulas can still be compared for the purpose of patch presence test. In the end, we use intra-function symbolic execution, *i.e.,* without following the callees (their return values will be made un-constrained as well), which in practice can already generate the formulas that make root instructions unique and stable.

(3) Symbolic execution in veritesting mode. Veritesting [47] is a technique that integrates static symbolic execution into dynamic symbolic execution to improve its efficiency. Dynamic symbolic execution is path-based, a same basic block belonging to multiple paths will be executed for multiple times, greatly increasing the overhead especially when there is a large number of paths. Static symbolic execution executes each basic block only once, but its formulas will be more complicated since it needs to carry different constraints of all paths that can reach current node. However, FIBER does not need to actually solve the formulas, instead, it only needs to compare these formulas extracted from reference and target binaries, thus, the formula complexity matters less for us. Note that this means an operand may sometimes have more than one formulas: consider when the true and false branch of a `if` statement merges. When we regard a binary signature as matched in the target, we require that the computed formulas in the target contain all of the formulas in the signature (could be a superset). If at least one formula is missing, we consider the corre-

sponding source code in the target to have missed certain important code that contributes to the signature.

5.6 Evaluation

In this section, we systematically evaluate FIBER for its effectiveness and efficiency.

Dataset. We choose Android kernels as our evaluation dataset. This is because Android is not only popular but also fragmented with many development branches maintained by different vendors such as Samsung and Huawei [102]. Although Google has open-sourced its Android kernels and maintained a frequently-undated security bulletin [4], other Android vendors may not port the security patches to their own kernels timely. Besides, even though required by open source license, many vendors choose not to open source their kernels or make it extremely inconvenient (with substantial delays and only periodic releases). This makes Android kernels an ideal target. We collect two kinds of dataset specifically:

(1) Reference kernel source code and security patches. We choose the open-source "angler" Android kernel (v3.10) used by Google's Nexus 6P as our reference. We then crawl the Android security bulletin from June 2016 to May 2017 and collect all published vulnerabilities related security patches[‖] for which we can locate the affected function(s) in the reference kernel image (*e.g.,* it may use a different driver than the one gets patched, or the affected function itself may be inlined). We also exclude one special patch that changes only a variable type in its declaration, requiring type inference at the binary level to handle,

[‖]Some security patches are not made publicly available on the Android Security Bulletin.

which we don't support currently as mentioned in §5.4.2.2. In total we collected 107 security patches that are applicable to our reference kernel.

(2) Target Android kernel images and source code. Besides the reference kernel, we also collect 8 Android kernel images from 3 different mainstream vendors with different timestamps and versions as listed in table 5.2. Note that vendors publish way more binary images (sometimes once every month) than the source code packages. We only evaluate the binary images for which we can find the corresponding source code, which serves only as ground truth of the patch presence test.

All our evaluations are performed on a server with Intel Xeon E5-2640 v2 CPU and 64 GB memory.

5.6.1 Experiment Procedure

To test patch presence in the target binary, we follow the steps below:

Reference binary preparation. As shown in Fig 5.2, we first need to compile the reference source code to binary, based on which we will generate the binary signatures. The availability of source code enables us to freely choose compilers, their options, and the target architecture. Naturally, we should choose the compilation configuration that is closest to the one used for target binary, which can maximize the accuracy. To probe the compilation configuration used for the target binary, we first compile multiple reference binaries with all combinations of common compilers (we use gcc and clang) and optimization levels (we use levels O1 - O3 and Os[**]), then use BinDiff [8] to test the similarity of each reference binary and the target binary, the most similar reference binary will finally be used for binary

[**]Optimize for size.

Device	No.	Patch Cnt*	Build Date (mm/dd/yy)	Kernel Version	Accuracy				Online Matching Time (s)			
					TP	TN	FP	FN	Total	Avg	~70%	Max.
Samsung S7	0	102	06/24/16	3.18.20	42	56	0	4(3.92%)	1690.43	16.57	8.47	306.47
	1	102	09/09/16	3.18.20	43	55	0	4(3.92%)	1888.06	18.51	8.24	438.76
	2	102	01/03/17	3.18.31	85	11	0	6(5.88%)	2421.44	23.74	5.49	1047.10
	3	102	05/18/17	3.18.31	92	4	0	6(5.88%)	1770.66	17.36	5.33	386.94
LG G5	4	103	05/27/16	3.18.20	32	65	0	6(5.88%)	2122.37	20.61	8.90	648.93
	5	103	10/26/17	3.18.31	95	0	0	8(7.77%)	1384.47	13.44	4.76	229.46
Huawei P9	6	31	02/22/16	3.10.90	10	20	0	1(3.23%)	390.35	12.59	8.47	89.35
	7	30	05/22/17	4.1.18	25	2	0	3(10.00%)	515.64	17.19	7.4	279.49

* Some patches we collected are not applicable for certain test subject kernels.

Table 5.2: Binary Patch Presence Test: Accuracy and Online Matching Performance

signature generation. Following this procedure (which is yet to be automated), we observed in our evaluation that kernel 6 and 7 as shown in table 5.2 use gcc with O2 optimization level, while all other 6 kernels use gcc with Os optimization level, which is confirmed by our inspection of the source code compilation configurations (*e.g.,* Makefile).

Offline signature generation and validation. For each security patch, we retain at most three binary signatures, after testing their uniqueness by matching them against both patched and un-patched reference kernel images. If nothing is unique, we will add more contexts to existing non-unique signatures.

Online matching. Given a specific security patch, we will try to match all its binary signatures in the target kernels. Note that all Android kernel images are compiled with symbol tables. We therefore can easily locate the affected functions. As long as one signature can be matched with a match count no less than that in reference patched kernel, we will say the patch exists in the target. As a performance optimization, we will first match the "fastest-to-match" signature.

5.6.2 Accuracy

We list the patch presence test results for target Android kernel images in table 5.2. It is worth noting that our patch collection is oriented to “angler” kernel, which will run on the Qualcomm hardware platform, while kernel 6 and 7 intend to run on a different platform (*i.e.,* Kirin), thus many device driver related patches do not apply for kernel 6 and 7 (we cannot even locate the same affected functions).

Overall, our accuracy is excellent. There are no false positives (FP) across the board and very few false negatives (FN). In patch presence test, we assume that all patches are not applied by default. It has to be proven otherwise. In practice, FP may lead developers to wrongly believe that a patch has been applied while in reality not (a serious security risk). In contrast, FN only costs some extra time for analysts to realize that the code is actually patched (or perhaps unaffected due to other reasons) while we say it is not. Thus, we believe FN is more tolerable than FP. Since we have no FP, we manually inspect each FN case to analyze the root causes:

(1) Function inline. Function inline behaviors may vary across different compilers and binaries. A same function may be inlined in some binaries but not others, or inlined in different ways. Some of our signatures (*e.g.,* the signature for `CVE-2016-8463`) model inline function calls based on the reference kernel image, if the target kernel has a different inline behavior, our signatures will fail to match. To address this problem, we need to generate binary signatures based on a collection of different kernel images to anticipate such behaviors.

(2) Function prototype change. Although rare, sometimes the function prototype will change across different kernel images. Specifically, the number and order of the function parameters may vary. As discussed in §5.4.3.1, we will differentiate the parameter order, thus, if a same parameter has different orders in reference and target kernels, the match will fail. We have one such case (`CVE-2014-9893`) in the evaluation. To solve this problem, we can extend our current implementation with techniques such as parameter profiling (see §5.4.3.1).

(3) Code customization. As discussed in §5.4.2, extra contexts are necessary if original patch change site is not unique. However, the contexts may be different across various kernel images due to code customization, although the patch change site remains the same. If this happens, our signature (with contexts extracted from the reference kernel) will not match, although the target kernel image has been patched. We encountered such a case in Samsung kernels for `CVE-2015-8942`. Such customizations are generally hard to anticipate and it will likely still cause a FN even if the source code of the target is given. This is why we prefer not to add contexts. If we can use more fine-grained binary analysis such as parameter and local variable profiling, we may be able to avoid using contexts.

(4) Patch adaptation. A patch may need to be adapted for kernels maintained by different vendors since the vulnerable functions are not always exactly the same across different kernel branches. Adaptation can also happen when a patch is back-ported to an older kernel version. In our evaluation, we find that this happens in some target images for `CVE-2016-5696`. Strictly speaking, FIBER intends to detect exactly the same patch as

appeared in the reference kernel, however, to be conservative, we still regard such cases as false negatives.

(5) Other engineering issues. Some FN cases are caused by engineering issues. For example, certain binary instructions cannot be recognized and decoded by the frontend of angr (two cases in total), which will affect the subsequent CFG generation and symbolic execution.

5.6.3 Performance

In this section we evaluate FIBER's runtime performance for both offline signature generation and online matching. We list the time consumption of the offline phase in table 5.3 and that of online phase in table 5.2. From the tables, we can see that a small fraction of patches needs much longer time to be matched than average, this is usually because the change sites in these patches are positioned in very large and complex functions (*e.g.,* `CVE-2017-0521`), thus the matching engine may encounter root instructions deep inside the function. However, most patches can be analyzed, translated and matched in a reasonable time. In the end, we argue that a human will take likely minutes, if not longer, to verify a patch anyways. An automated and accurate solution like ours is still preferable, not to mention that we can parallelize the analysis of different patches.

5.6.4 Unported Patches

As shown in table 5.2, for all the test subjects except kernel #5, FIBER produces some TN cases, which suggests un-patched vulnerabilities. If related security patches had

Step	Total	Cnt. **	Avg.	~70%
Analyze	21.52s	107	0.20s	-
Translation	1608.52s	293	5.49s	6.29s
Match Ref.0 *	2647.78s	293	9.04s	6.00s
Match Ref.1 *	3415.54s	293	11.66s	7.56s

* Match against reference kernels for uniqueness test.
* 0 for un-patched kernel, 1 for patched kernel.
** Analyze: Patch. Others: Binary Signature.

Table 5.3: Offline Phase Performance

CVE	Patch Date * (mm/yy)	Type**	Severity*
CVE-2014-9781	07/16	P	High
CVE-2016-2502	07/16	P	High
CVE-2016-3813	07/16	I	Moderate
CVE-2016-4578	08/16	I	Moderate
CVE-2016-2184	11/16	P	Critical
CVE-2016-7910	11/16	P	Critical
CVE-2016-8413	03/17	I	Moderate
CVE-2016-10200	03/17	P	Critical
CVE-2016-10229	04/17	E	Critical

* Obtained from Android security bulletin.
** **P:** Privilege Elevation **E:** Remote Code Execution
** **I:** Information Disclosure

Table 5.4: Potential Security Loopholes

already been available before the test subject's release date, then it means that the test subject fails to apply the patch timely. Table 5.4 lists all the vulnerabilities whose patches fail to be propagated to one or multiple test subject kernel(s) timely in our evaluation. Note that for security concerns, we do not correlate these vulnerabilities with actual kernels in table 5.2.

From table 5.4, we can see that even some critical vulnerabilities were not patched in time, indicating a good potential that they can be leveraged to compromise the kernel entirely to execute arbitrary code. One such case is a patch delayed for more than half a year affecting a major vendor (who confirmed the case and requested to be anonymized). This illustrates the value of tools like FIBER.

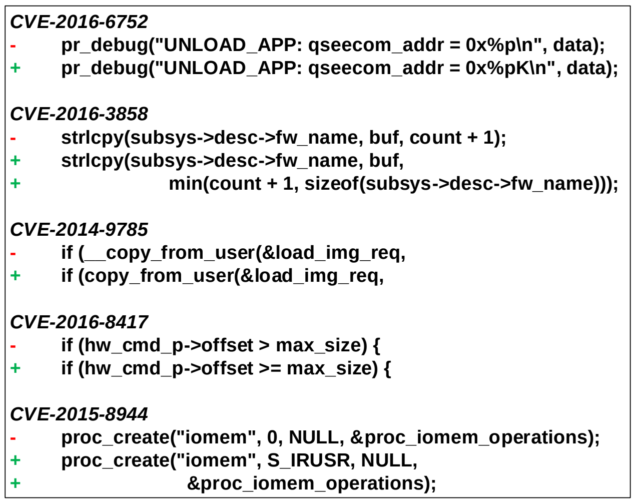
Besides, we also identify 4 vulnerabilities in table 5.4 that eventually got patched in a later kernel release but not in the earliest kernel release after the patch release date, indicating a significant delay of the patch propagation process.

It is worth noting that FIBER intends to test whether the patch exists in the target kernel, however, the absence of a security patch does not necessarily mean that the target kernel is exploitable. So the further verification is still needed.

5.6.5 Case Study

In this section, we demonstrate some representative security patches used in our evaluation to show the strength of FIBER compared to other solutions.

Format String Change. There are 5 patches in our collection that intend to change only the format strings as function arguments. Take the patch for `CVE-2016-6752` in Fig 5.6 as an example, the specifier `p` is changed to `pK`. It will be impossible to detect it at binary level without dereferencing the string pointer, since all other features (*e.g.,*. topology, instruction type.) remain exactly the same. However, without patch insights, it is extremely difficult to decide which register or memory location should be regarded as a pointer and whether it should be dereferenced in the matching process, rendering all binary-only solutions ineffective in this case. While FIBER can correctly decide that the only thing changed is the



```
CVE-2016-6752
-       pr_debug("UNLOAD_APP: qseecom_addr = 0x%p\n", data);
+       pr_debug("UNLOAD_APP: qseecom_addr = 0x%pK\n", data);

CVE-2016-3858
-       strlcpy(subsys->desc->fw_name, buf, count + 1);
+       strlcpy(subsys->desc->fw_name, buf,
+                       min(count + 1, sizeof(subsys->desc->fw_name)));

CVE-2014-9785
-       if (__copy_from_user(&load_img_req,
+       if (copy_from_user(&load_img_req,

CVE-2016-8417
-       if (hw_cmd_p->offset > max_size) {
+       if (hw_cmd_p->offset >= max_size) {

CVE-2015-8944
-       proc_create("iomem", 0, NULL, &proc_iomem_operations);
+       proc_create("iomem", S_IRUSR, NULL,
+                       &proc_iomem_operations);
```

Figure 5.6: Example Security Patches

argument format string (see §5.4.2) and then test patch presence by matching the string content.

Small Change Site. It is very common that a security patch will only introduce small and subtle changes, such as the one for `CVE-2016-8417` shown in Fig 5.6, where the operator ">" is replaced with ">=". Such a change has no impact on the CFG topology and only one conditional jump instruction will be slightly different. Thus, it will be extremely difficult to differentiate the patched and un-patched functions without the fine-grained signature. FIBER handles this case correctly because the conditional jump is part of the root instruction and we will check the comparison operator associated with it.

Patch Backport. A downstream kernel may selectively apply patches (security or other bug fixes), which can cause functions to look different from upstream. Our reference kernel (v3.10) is actually a downstream compared to all test subjects except #6 as shown in table 5.2. The patch for `CVE-2016-3858` (shown in Fig 5.6) has a prior patch in the upstream (which deletes a "if-then-return" statement) for the same affected function, which was not applied to our reference kernel, making the two functions look different although both patched. FIBER is robust to such backporting cases because the generated binary signature is fine-grained and related to only a single patch.

Multiple Patched Function Versions. After a security patch is applied, the same function may be modified by future patches as well. Thus, similar to the backporting cases, two patched functions can still be different because they are on different versions. `CVE-2014-9785` is such an example. FIBER can still precisely locate the same change site as shown in Fig 5.6 even when faced with a much newer target function, which differs significantly with the reference function.

Constant Change. Patch for `CVE-2015-8944` in Fig 5.6 only changes a function argument from 0 to a pre-defined constant `S_IRUSR` (0x100 in reference kernel). Once again, such a small change makes the patched and un-patched functions highly similar. Even though a solution wants to strictly differentiate constant values, it is in general unsafe because the constants are prone to change across binaries. However, with the insights of the fine-grained change site, FIBER can correctly figure out that only the value of the 2nd function argument matters in the matching and it should be non-zero if patched, thus effectively handle such cases.

Similar Basic Blocks. FIBER generates fine-grained signatures containing only a limited set of basic blocks (see §5.4.3.1). It is likely that there will be other similar basic blocks as the signature if we only look at the basic block level semantics. One such example has been shown in Fig 5.1 and discussed in §5.3. Previous work based on basic block level semantics [106, 105] may fail to handle such cases, While FIBER tries to integrate function level semantics into the local CFG, resulting in fine-grained signatures that are both stable and unique.

5.7 Conclusion

In this chapter, we formulate a new problem of patch presence test under "source to binary" scenario. We then design and implement FIBER, a fully automatic solution which can take the best advantage of source level information for accurate and precise patch presence test in binaries. FIBER has been systematically evaluated with real-world security patches and a diverse set of Android kernel images, the results show that it can achieve an excellent accuracy with acceptable performance, thus highly practical for security analysts.

Chapter 6

Conclusion and Future Work

In this thesis we explore security problems and mitigation for the kernel and its ecosystem from the perspective of its development cycle. We first pinpoint vulnerabilities in the kernel design and implementation phase, within the Android kernel ION memory management subsystem, for the identified issues, we also develop tools or discuss potential solutions to eliminate them. Then to facilitate a better kernel testing for the stealthy high-order taint style vulnerabilities, we design and implement a novel static analysis tool SUTURE, to effectively and efficiently detect such vulnerabilities. To study the attacks for the deployed kernels, we conduct a systematic study on the Android one-click-root apps, figuring out their workflows and extracting their internal exploits by reverse engineering, we further compare these provider exploits with the publicly available ones, our results warn the community of the high risks of abusing such one-click-root apps. Finally, to improve the maintenance of the kernel ecosystem, we propose a precise and accurate patch presence test tool FIBER on the binary level, with FIBER, the defenders can easily pinpoint the unfixed vulnerabilities in the closed-source kernel images and then apply the related patches in time, effectively reducing the attack windows.

We believe that the development cycle provides us with a more holistic view to study the kernel security problems, we hope that by inspecting the security issues throughout the development pipeline, the kernel ecosystem can be secured in a more comprehensive way. That being said, there are still many open problems along this line and we treat them as the potential future work of this thesis, more specifically, we list some of them as following:

For the kernel design and implementation: This thesis studies the ION memory management subsystem, however, the kernel contains many other different subsystems (*e.g.,* the file system, the sound driver, *etc.*) and the new functionalities are added to the kernel on a daily basis. It is necessary to perform a systematic screening to various kernel modules to identify the potential design and implementation flaws from the security perspective (*e.g.,* many subsystems are not designed with security in mind).

For the kernel testing: One direction is to formulate more stealthy and traditionally difficult-to-detect security vulnerabilities and then try to develop tools to automatically detect them in the kernel testing phase (*e.g.,* the high-order taint vulnerability is one class of them), another direction is to refine the current kernel testing techniques, *e.g.,* static analysis is generally more comprehensive but prone to false alarms, while dynamic testing is accurate but cannot guarantee the code coverage, both of them still have a large room for improvement, and to combine them together is also a promising direction.

For the kernel deployment: We study the privilege escalation attacks for the deployed Android kernels by inspecting the one-click root apps in this thesis, but in practice, there exists more types of attacks (*e.g.,* denial-of-service, information leakage, *etc.*) which are also worth studying. Besides, how to harden the deployed kernels to defend against these attacks is also an important research topic.

For the kernel maintenance: FIBER is able to test the patch presence in the binary kernels with a high accuracy, however, there is still room for improvements, *e.g.,* the different compiler or kernel config options can still affect the signature matching, we are planning to re-design the signature format to make it more resistent to the syntax change. In a bigger picture, the patch presence test is only one step at the very beginning in the patch propagation process, while there still exists many problems in this process, *e.g.,* the original patch may be unsuitable for a customized downstream kernel in the ecosystem, in this situation how can we automatically adapt the patch? We list these open problems as our future work.

Bibliography

[1] https://sites.google.com/a/androidionhackdemo.net/androidionhackdemo/.

[2] Android Low Memory Killer. https://android.googlesource.com/kernel/common.git/+/android-3.4/drivers/staging/android/lowmemorykiller.c.

[3] Android MediaServer Bug Traps Phones in Endless Reboots. http://blog.trendmicro.com/trendlabs-security-intelligence/android-mediaserver-bug-traps-phones-in-endless-reboots/.

[4] Android Security Bulletin. https://source.android.com/security/bulletin/.

[5] Android Vulnerabilities – All vulnerabilities. http://androidvulnerabilities.org/all.html.

[6] Arm TrustZone Technology. http://www.arm.com/products/processors/technologies/trustzone/.

[7] Beating up on Android. http://titanium.immunityinc.com/infiltrate/archives/Android_Attacks.pdf.

[8] BinDiff. https://www.zynamics.com/bindiff.html.

[9] Contagio minidump. http://contagiominidump.blogspot.com.

[10] CVE-2014-3153 aka towelroot. https://github.com/timwr/CVE-2014-3153.

[11] CVE-2016-2068. https://nvd.nist.gov/vuln/detail/CVE-2016-2068.

[12] CVE-2017-0608. https://nvd.nist.gov/vuln/detail/CVE-2017-0608.

[13] CVE: Vulnerabilities By Year. https://www.cvedetails.com/browse-by-date.php.

[14] Device Tree. https://www.kernel.org/doc/Documentation/devicetree/usage-model.txt.

[15] Don't Root Robots: Breaks in Google's Android Platform. https://jon.oberheide.org/files/bsides11-dontrootrobots.pdf.

[16] Exploit DB database. https://exploit-db.com/.

[17] Github Annual Report. https://octoverse.github.com/.

[18] How To Root An AT&T HTC One X. http://rootzwiki.com/topic/26320-how-to-root-an-att-htc-one-x-this-exploit-supports-185/.

[19] Integer Overflow leading to Heap Corruption while Unflattening GraphicBuffer. http://seclists.org/fulldisclosure/2015/Mar/63.

[20] iRoot, Retrieved on May 10, 2015. http://www.mgyun.com/m/en.

[21] It's Bugs All the Way Down. http://vulnfactory.org/.

[22] Linux in 2020. https://www.linux.com/news/linux-in-2020-27-8-million-lines-of-code-in-the-kernel-1-3-million-in-systemd/.

[23] NetworkX Python Package. https://networkx.github.io/.

[24] One Click Root for Android, Retrieved on May 10, 2015. http://www.oneclickroot.com/.

[25] Rage Against the Cage. http://stealth.openwall.net/xSports/RageAgainstTheCage.tgz.

[26] Razr Blade Root. http://vulnfactory.org/public/razr_blade.zip.

[27] Root Genius, Retrieved on May 10, 2015. http://www.shuame.com/en/root/.

[28] Root the Droid 3. http://vulnfactory.org/blog/2011/08/25/rooting-the-droid-3/.

[29] [Root] ZTE z990g Merit (An avail variant). http://forum.xda-developers.com/showthread.php?t=1714299.

[30] [Root/Write Protection Bypass] MotoX (no unlock needed). http://forum.xda-developers.com/moto-x/orig-development/root-write-protection-bypass-motox-t2444957.

[31] Samsung Knox. https://www.samsungknox.com/.

[32] Security Patch for CVE-2015-8955. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git/commit/?id=8fff105e13041e49b82f92eef034f363a6b1c071.

[33] STAC - Static Taint Analysis for C. http://code.google.com/p/tanalysis/.

[34] TacoRoot. https://github.com/CunningLogic/TacoRoot.

[35] The Linux Kernel. https://www.kernel.org/.

[36] The LLVM Compiler Infrastructure. https://llvm.org/.

[37] Virus Profile: Exploit/MempoDroid.B. `http://home.mcafee.com/virusinfo/virusprofile.aspx?key=1003986`.

[38] VirusTotal. `https://www.virustotal.com/`.

[39] Xoom FE: Stupid Bugs, and More Plagiarism. `http://vulnfactory.org/blog/2012/02/18/xoom-fe-stupid-bugs-and-more-plagiarism/`.

[40] The Android ION memory allocator. `https://lwn.net/Articles/480055/`, 2012.

[41] Patch: sparc32: dma_alloc_coherent must honour GFP_ZERO. `https://patchwork.ozlabs.org/patch/386217/`, 2014.

[42] Yousra Aafer, Xiao Zhang, and Wenliang Du. Harvesting Inconsistent Security Configurations in Custom Android ROMs via Differential Analysis. In *USENIX SECURITY*, 2016.

[43] A. ARMANDO, A. MERLO, M. MIGLIARDI, , and L VERDERAME. Would You Mind Forking this Process? A Denial of Service Attack on Android (and Some Countermeasures). In *Information S&P Research*, 2016.

[44] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *NDSS*, 2014.

[45] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *PLDI*, 2014.

[46] A. Averbuch, M. Kiperberg, and N.J. Zaidenberg. Truly-Protect: An Efficient VM-Based Software Protection. *Systems Journal, IEEE*, 2013.

[47] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. ICSE'14.

[48] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. Efficient and flexible discovery of php application vulnerabilities. EuroS&P'17.

[49] Brenda S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM J. Comput.*, 26(5):1343–1362, October 1997.

[50] Martial Bourquin, Andy King, and Edward Robbins. Binslayer: Accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop.*

[51] Fraser Brown, Deian Stefan, and Dawson Engler. Sys: a static/symbolic tool for finding good bugs in good (browser) code. Usenix Security'20.

[52] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. On the Limits of Information Flow Techniques for Malware Analysis and Containment. In *DIMVA*, 2008.

[53] Qi Alfred Chen, Zhiyun Qian, Yunhan Jack Jia, Yuru Shao, and Zhuoqing Morley Mao. Static detection of packet injection vulnerabilities: A case for identifying attacker-controlled implicit information leaks. CCS'15.

[54] Qi Alfred Chen, Zhiyun Qian, and Z. Morley Mao. Peeking into Your App Without Actually Seeing It: UI State Inference and Novel Android Attacks. In *Proc. of USENIX Security*, 2014.

[55] Kai Cheng, Qiang Li, Lei Wang, Qian Chen, Yaowen Zheng, Limin Sun, and Zhenkai Liang. Dtaint: detecting the taint-style vulnerability in embedded device firmware. DSN'18.

[56] Fred Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich. Effective representation of aliases and indirect memory operations in ssa form. In *International Conference on Compiler Construction*, pages 253–267. Springer, 1996.

[57] Zheng Leong Chua, Yanhao Wang, Teodora Baluta, Prateek Saxena, Zhenkai Liang, and Purui Su. One engine to serve'em all: Inferring taint rules without architectural semantics. NDSS'19.

[58] Christian Collberg, Clark Thomborson, and Douglas Low. A Taxonomy of Obfuscating Transformations. Technical report, The University of Auckland, 1997.

[59] Christian S. Collberg and Clark Thomborson. Watermarking, Tamper-proffing, and Obfuscation: Tools for Software Protection. *IEEE Trans. Softw. Eng.*, 2002.

[60] Marco Cova, Viktoria Felmetsger, Greg Banks, and Giovanni Vigna. Static detection of vulnerabilities in x86 executables. ACSAC'06.

[61] Weidong Cui, Marcus Peinado, Zhilei Xu, and Ellick Chan. Tracking rootkit footprints with a practical memory analysis system. USENIX Security'12.

[62] Johannes Dahse and Thorsten Holz. Static detection of second-order vulnerabilities in web applications. Usenix Security'14.

[63] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[64] Niall Douglas. User Mode Memory Page Allocation: A Silver Bullet For Memory Allocation? Technical report, 2011.

[65] Joshua J. Drake, Zach Lanier, Collin Mulliner, Pau Oliva Fora, Stephen A. Ridley, and Georg Wicherski. *Android Hacker's Handbook*. Wiley, 2014.

[66] Nicolas Falliere, Liam O Murchu, and Eric Chien. W32.Stuxnet Dossier. Technical report, Symanetic, 2011.

[67] Qian Feng, Minghua Wang, Mu Zhang, Rundong Zhou, Andrew Henderson, and Heng Yin. Extracting conditional formulas for cross-platform bug search. ASIACCS'17.

[68] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. CCS '16.

[69] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. FSE'14.

[70] Debin Gao, Michael K. Reiter, and Dawn Song. Binhunt: Automatically finding semantic differences in binary programs. In *Information and Communications Security*, 2008.

[71] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *TRUST*, 2012.

[72] Michael I Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. Information-flow Analysis of Android Applications in DroidSafe. In *NDSS*, 2015.

[73] Ivan Gotovchits, Rijnard Van Tonder, and David Brumley. Saluki: finding taint-style vulnerabilities with static property checking. In *Proceedings of the NDSS Workshop on Binary Analysis Research*, 2018.

[74] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *NDSS*, 2012.

[75] Neville Grech and Yannis Smaragdakis. P/taint: unified points-to and taint analysis.

[76] Dan Guido and Mike Arpaia. *The Mobile Exploit Intelligence Project*. Blackhat EU, 2012.

[77] You Joung Ham, Won-Bin Choi, and Hyung-Woo Lee. Mobile Root Exploit Detection based on System Events Extracted from Android Platform. In *SAM*, 2013.

[78] Xiali Hei, Xiaojiang Du, and Shan Lin. Two Vulnerabilities in Android OS Kernel. In *ICC*, 2013.

[79] Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. Machine-learning-guided selectively unsound static analysis. ICSE'17.

[80] Heqing Huang, Sencun Zhu, Kai Chen, and Peng Liu. From System Services Freezing to System Server Shutdown in Android: All You Need Is a Loop in an App. In *CCS*, 2015.

[81] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. Scalable and precise taint analysis for android. ISSTA'15.

[82] Frank K Hwang, Dana S Richards, and Pawel Winter. *The Steiner tree problem*, volume 53. Elsevier, 1992.

[83] Jiyong Jang, Abeer Agrawal, and David Brumley. Redebug: finding unpatched code clones in entire os distributions. Oakland'12.

[84] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. ICSE'07.

[85] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. Oakland'06.

[86] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 27–36, 2006.

[87] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, Jul 2002.

[88] W. M. Khoo, A. Mycroft, and R. Anderson. Rendezvous: A search engine for binary code. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013.

[89] S. Kim, S. Woo, H. Lee, and H. Oh. Vuddy: A scalable approach for vulnerable code clone discovery. Oakland'17.

[90] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static Disassembly of Obfuscated Binaries. In *Proc. of USENIX Security Symposium*, 2004.

[91] JongHyup Lee, Thanassis Avgerinos, and David Brumley. Tie: Principled reverse engineering of types in binary programs. NDSS'11.

[92] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities. In *Oakland*, 2014.

[93] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, March 2006.

[94] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. Vulpecker: an automated vulnerability detection system based on code similarity analysis. ACSAC'16.

[95] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *BADGERS*, 2014.

[96] Cullen Linn and Saumya Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *ACM CCS*, 2003.

[97] Kangjie Lu, Zhichun Li, Vasileios Kemerlis, Zhenyu Wu, Long Lu, Cong Zheng, Zhiyun Qian, Wenke Lee, and Guofei Jiang. Checking More and Alerting Less: Detecting Privacy Leakages via Enhanced Data-flow Analysis and Peer Voting. In *NDSS*, 2015.

[98] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *CCS*, 2012.

[99] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. {DR}.{CHECKER}: A soundy analysis for linux kernel drivers. Usenix Security'17.

[100] Jiang Ming, Meng Pan, and Debin Gao. ibinhunt: Binary hunting with inter-procedural control flow. In *Proceedings of the 15th International Conference on Information Security and Cryptology.*

[101] Michal Nazarewicz. A Deep Dive into CMA. `https://lwn.net/Articles/486301/`.

[102] OpenSignal. Android Fragmentation Visualized. `https://opensignal.com/reports/2015/08/android-fragmentation/`.

[103] OpenSignal. Android Fragmentation Visualized. `http://opensignal.com/reports/2015/08/android-fragmentation/`, 2015.

[104] Yeongung Park, ChoongHyun Lee, Chanhee Lee, JiHyeog Lim, Sangchul Han, Minkyu Park, and Seong-Je Cho. RGBDroid: A Novel Response-Based Approach to Android Privilege Escalation Attacks. In *LEET*, 2012.

[105] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-architecture bug search in binary executables. Oakland'15.

[106] J. Pewny, F. Schuster, C. Rossow, L. Bernhard, and T. Holz. Leveraging semantic signatures for bug search in binary programs. ACSAC'14.

[107] David A. Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. USENIX Security'15.

[108] Rolf Rolles. Unpacking Virtualization Obfuscators. In *WOOT*, 2009.

[109] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27, 1988.

[110] Brendan Saltaformaggio, Rohit Bhatia, Zhongshu Gu, Xiangyu Zhang, and Dongyan Xu. GUITAR: Piecing Together Android App GUIs from Memory Images. In *CCS*, 2015.

[111] Yuru Shao, Jason Ott, Qi Alfred Chen, Zhiyun Qian, and Z Morley Mao. Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework. In *NDSS*, 2016.

[112] Bhargava Shastry, Federico Maggi, Fabian Yamaguchi, Konrad Rieck, and Jean-Pierre Seifert. Static exploration of taint-style vulnerabilities found by fuzzing. WOOT'17.

[113] Dongdong She, Yizheng Chen, Abhishek Shah, Baishakhi Ray, and Suman Jana. Neutaint: Efficient dynamic taint analysis with neural networks. Oakland'20.

[114] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. Oakland'16.

[115] Stephen Smalley and Robert Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *NDSS*, 2013.

[116] Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. F4f: taint analysis of framework-based web applications. OOPSLA'11.

[117] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266. ACM, 2016.

[118] Yuan Tian, Julia Lawall, and David Lo. Identifying Linux bug fixing patches. ICSE'12.

[119] Jacob I. Torrey. HARES: Hardened Anti-Reverse Engineering System. Technical report, Assured Information Security, Inc., 2015.

[120] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. Taj: effective taint analysis of web applications.

[121] Tielei Wang, Yeongjin Jang, Yizheng Chen, Simon Chung, Billy Lau, and Wenke Lee. On the Feasibility of Large-Scale Infections of iOS Devices. In *Usenix Security'14*.

[122] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. Improving integer security for systems with {KINT}. OSDI'12.

[123] Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. ICSE'08.

[124] Fengguo Wei, Sankardas Roy, Xinming Ou, et al. Amandroid: A Precise and General Inter-Component Data Flow Analysis Framework for Security Vetting of Android Apps. In *CCS*, 2014.

[125] Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. The Impact of Vendor Customizations on Android Security. In *CCS*, 2013.

[126] Wen Xu. *Ah! Universal Android Rooting is Back.* Blackhat, 2015.

[127] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. CCS '17.

[128] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. Automatic inference of search patterns for taint-style vulnerabilities. Oakland'15.

[129] Junyuan Zeng, Yangchun Fu, Kenneth A. Miller, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Obfuscation Resilient Binary Code Reuse Through Trace-oriented Programming. In *ACM CCS*, 2013.

[130] Yizhuo Zhai, Yu Hao, Hang Zhang, Daimeng Wang, Chengyu Song, Zhiyun Qian, Mohsen Lesani, Srikanth V Krishnamurthy, and Paul Yu. Ubitect: a precise and scalable method to detect use-before-initialization bugs in linux kernel. FSE'20.

[131] Hang Zhang, Dongdong She, and Zhiyun Qian. Android ion hazard: The curse of customizable memory management system. CCS'16.

[132] Hang Zhang, Dongdong She, and Zhiyun Qian. Android Root and Its Providers: A Double-Edged Sword. In *CCS*, 2015.

[133] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M Azab, and Ruowen Wang. Pex: A permission check analysis framework for linux kernel. Usenix Security'19.

[134] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. The Peril of Fragmentation: Security Hazards in Android Device Driver Customizations. In *Oakland*, 2014.

[135] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. The Peril of Fragmentation: Security Hazards in Android Device Driver Customizations. In *IEEE Security and Privacy*, 2014.

[136] Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. In *IEEE Security and Privacy*, 2012.

[137] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *NDSS*, 2012.