UC Irvine
UC Irvine Electronic Theses and Dissertations

Title
Securing Mobile Devices Through Discovery, Mitigation, and Prevention of Vulnerabilities

Permalink
https://escholarship.org/uc/item/6g64s2jr

Author
Seyed Talebi, Seyed Mohammadjavad

Publication Date
2022

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA,
IRVINE

Securing Mobile Devices Through Discovery, Mitigation, and Prevention of Vulnerabilities

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Seyed Mohammadjavad Seyed Talebi

Dissertation Committee:
Professor Ardalan Amiri Sani, Chair
Professor Gene Tsudik
Professor Qi Alfred Chen

2022

Portion of Chapter 2 © 2018 USENIX, reprinted, with permission, from [214]
Portion of Chapter 3 © 2021 USENIX, reprinted, with permission, from [215]
Portion of Chapter 4 © 2021 ACM, reprinted, with permission, from [213]
All other materials © 2022 Seyed Mohammadjavad Seyed Talebi

DEDICATION

This thesis work is dedicated to my lovely wife, Saba, who has always been a constant source of support and encouragement for me. I am truly thankful for having you in my life.

Contents

Page

LIST OF FIGURES **vi**

LIST OF TABLES **viii**

ACKNOWLEDGMENTS **ix**

VITA **x**

ABSTRACT OF THE DISSERTATION **xii**

1 Introduction **1**
1.1 Vulnerability Discovery 3
1.2 Vulnerability Mitigation 5
1.2.1 Bowknots 6
1.2.2 MegaMind 7
1.3 Vulnerability Prevention 8

2 Charm: Facilitating Dynamic Analysis of Device Drivers of Mobile Systems **10**
2.1 Motivation 14
2.1.1 Manual Interactive Debugging 14
2.1.2 Record-and-Replay 14
2.1.3 Fuzzing 15
2.2 Remote Device Driver Execution 17
2.2.1 Device and Device Driver Interactions 18
2.2.2 Device Driver Initialization 19
2.2.3 Low-Latency USB Channel 20
2.2.4 Dependencies 21
2.2.5 Porting a Device Driver to Charm 22
2.3 Implementation & Prototype 24
2.4 Evaluation 25
2.4.1 Engineering Effort 26
2.4.2 Performance 26
2.4.3 Record-and-Replay 28
2.4.4 Bug Finding 29

3 Undo Workarounds for Kernel Bugs **31**
3.1 Motivation . . . 34
3.1.1 Unpatched Kernel Bugs . . . 34
3.1.2 Problems with Unpatched Kernel Bugs . . . 35
3.1.3 Current Approaches . . . 36
3.2 Overview . . . 37
3.2.1 Goals . . . 37
3.2.2 Key Idea & Design . . . 38
3.2.3 Workflow . . . 40
3.3 Bowknots . . . 41
3.3.1 Function Instrumentation . . . 42
3.3.2 Recursive Undo of Call Stack . . . 46
3.4 Automatic Generation of Bowknots . . . 47
3.4.1 Function-Pair Knowledge Database . . . 48
3.4.2 Generating the Undo Block . . . 50
3.4.3 Incompleteness and Confidence Score . . . 52
3.5 Implementation . . . 53
3.6 Evaluation . . . 56
3.6.1 Effectiveness . . . 56
3.6.2 Manual Effort for the Pair Database . . . 61
3.6.3 Performance Overhead . . . 62
3.6.4 Use-Case Evaluation . . . 62
3.7 Other Limitations . . . 66

4 MegaMind: A Platform for Security & Privacy Extensions for Voice Assistants **68**
4.1 Motivating Extensions . . . 72
4.2 Architectural Overview . . . 74
4.3 Trust & Threat Model . . . 75
4.4 Permission Enforcement . . . 78
4.4.1 Access Permissions . . . 79
4.4.2 Modification Permissions . . . 82
4.5 Non-interference . . . 83
4.5.1 Non-interference definition . . . 84
4.5.2 Non-interference guarantee . . . 85
4.6 Novel Security Features . . . 88
4.6.1 Secure Conversation . . . 88
4.6.2 Anonymous Query . . . 90
4.7 Implementation . . . 91
4.7.1 Key Implementation Components . . . 91
4.7.2 Performance Optimizations . . . 93
4.8 Evaluation . . . 94
4.8.1 Performance . . . 95
4.8.2 Effectiveness . . . 98

5 Split-Trust Machine Model 103
5.1 Background . . . 105
5.1.1 Trust Definitions . . . 105
5.1.2 Trust in Existing Systems . . . 107
5.2 Key Goal and Principle . . . 109
5.3 Split-Trust Machine Model . . . 110
5.3.1 Static Partitioning and Physical Isolation . . . 111
5.3.2 Exclusive Inter-Domain Communication . . . 112
5.3.3 Power Management . . . 114
5.3.4 Hardware Root of Trust . . . 114
5.3.5 High Performance I/O . . . 115
5.3.6 Domain and Mailbox Reset . . . 116
5.4 Prototype . . . 117
5.4.1 Verified Hardware Design . . . 118
5.5 Evaluation . . . 119
5.5.1 Hardware Cost . . . 119
5.5.2 Performance . . . 120

6 Related Work 123
6.1 Vulnerability Discovery . . . 123
6.1.1 Remote I/O Access . . . 123
6.1.2 Analysis of System Software . . . 124
6.1.3 Mobile Testing . . . 126
6.2 Vulnerability Mitigation . . . 127
6.2.1 Bug workarounds. . . . 127
6.2.2 Automatic fault recovery. . . . 127
6.2.3 Input filtering. . . . 129
6.2.4 Automated patching. . . . 129
6.2.5 Error handling analysis. . . . 130
6.2.6 Voice assistants security. . . . 131
6.3 Vulnerability Prevention . . . 133
6.3.1 Security by physical isolation. . . . 133
6.3.2 Secure I/O for TEEs. . . . 134
6.3.3 Time protection. . . . 134
6.3.4 Other TEE solutions. . . . 135

7 Conclusions 136

Bibliography 138

Appendix A Bugs description 157

List of Figures

Page

1.1 *Repetitive reboots when fuzzing the camera device driver of Nexus 5X.* 4

2.1 Charm enables a security analyst to run a mobile I/O device driver in a virtual machine and apply dynamic analysis to it. 12
2.2 (a) Device driver execution in a mobile system. (b) Remote device driver execution in Charm. 18
2.3 (a) Execution speed of the fuzzer. (b) Coverage of the fuzzer. 27

3.1 *High-level idea behind bowknots and Hecaton.* 39
3.2 *Example function in the Qualcomm KGSL GPU device driver after instrumentation with a bowknot. (Up) Automatically-triggered, (Down) Manually-triggered bowknot. The blue and bold text highlights the automatically added code. The green and italic text highlights the manually added lines. The code presented here is slightly modified from the actual function code and from the one generated by Hecaton for better readability.* 43
3.3 *Hecaton Confidence score prediction for Tuning and Testing sets* 61
3.4 *GPU and TCP performance as the number of executed bowknots increase. (a) Pixel3 GPU , (b) Pixel3 TCP, (c) x86 upstream Linux (running in QEMU) TCP.* 63
3.5 *The setup used in our fuzzing experiments.* 63
3.6 *(a) Total executed fuzzing programs. (b) Covered basic blocks (code coverage percentage is also reported on top of each bar).* 64
3.7 *Time taken for the fuzzer to discover a bug (i.e., trigger a bug for the first time). Each x-axis tick represents a unique bug.The points with no error bars represent bugs only found once during experiments* 65

4.1 *Amazon Alexa voice assistant architecture.* 73
4.2 *Adding the MegaMind extensibility platform to Amazon Alexa. MegaMind's functionality is shown in green.* 73
4.3 *Trigger rules description language.* 81
4.4 *Latency breakdown for different extensions for three platforms. For each extension, first, second and third bars, respectively, show the average latency for first, middle, and overall commands in a session. The last bar shows the baseline latency for that extension.* 95

4.5 *Impact of number of extensions on latency.* 97
4.6 *Extensions CPU utilization. For each extension, first, second, and third bar groups, respectively, represent laptop, RPi 4, and RPi 3.* 98

5.1 *(a) Traditional design where the OS isolates security-critical programs from normal programs. (b) Use of a TEE to isolate a security-critical program.* . . . 107
5.2 *Simplified overview of the split-trust machine model. The figure does not show all mailboxes for clarity.* 111
5.3 *Mailbox design.* 113

List of Tables

Page

2.1 Device drivers currently supported in Charm. 24
2.2 Bugs we found in device drivers through fuzzing with Charm. MI and LC refer to confirming the bug by Manual Inspection and by checking the driver's Latest Commits, respectively. 29

3.1 *CVEs and real kernel bugs tested with bowknots. (* In these cases, the system was functional right after mitigation by Talos, but it stopped working after a while due to a memory leak resulting from code disabling, **Bug1: bug in msm_camera_power_down)* 58
3.2 *Unpatched bugs experiments (x86 Linux kernel bugs reported by Syzbot). (*Average # of added undo statements for incomplete bowknots by Hecaton)* 58
3.3 *Bug injection experiments (camera device driver and Binder IPC).(*Average # of added undo statements for incomplete bowknots by Hecaton)* 59
3.4 *Effective fuzzing time. U. and B. refer to using unmodified kernel vs. a kernel updated with bowknots. The number of reboots are per hour. Up time which is the overall time during which the fuzzer is running including wasted reboot time is 24h for all experiments. Fuzz time (i.e., effective fuzz time) is the time during which the fuzzer is actually fuzzing the kernel of the device.* 59
3.5 *Bowknots vs. code disabling (Talos) for fuzzing.* 66

4.1 *MegaMind protection for attacks.* 78
4.2 *This table summarizes all possible types of interference extension E1 can cause on extension E2's execution. "✓" means MegaMind can prevent interference. In each case, interference is avoided by: Extensions' definition (E), Order of execution (O), Limitations on extensions (L), and Trust model (T).* 82
4.3 *MegaMind's detection errors. FN stands for false negatives, and FP for false positives.* 100

5.1 *Theorems we prove for our mailbox. Proving some of these theorems require proving multiple lemmas not listed here.* 118
5.2 *Cost of additional hardware in our machine.* 120
5.3 *Mailbox performance.* 121
5.4 *Storage performance.* 121
5.5 *Network performance.* 122

ACKNOWLEDGMENTS

I am incredibly thankful to my advisor, professor Ardalan Amiri Sani. During my PhD, he gave me the courage to work on challenging research problems, and with his patience, he taught me not to give up when I face obstacles. Without his knowledge and guidance, I could not finish my PhD.

I am also thankful to my committee members, professor Gene Tsudik and professor Qi Alfred Chen who dedicated their valuable time to give me their insightful feedback on this dissertation.

I would like to express my gratitude to my wonderful collaborators, professor Zhiyn Qian, Dr. Daniel Austin, Dr. Alec Wolman, and Dr. Stefan Saroiu. They always guided me towards the best research directions with their insightful ideas.

I would like to especially thank my friend, collaborator, and groupmate at TrussLab, Zhihao (Zephyr) Yao who co-led the split-trust machine project with me. Without his extraordinary effort to build OctopOS, a brand-new operating system to manage the split-trust machine, it was impossible to evaluate and show the effectiveness of my hardware design.

I am deeply thankful for my amazing wife and my family who have always been supportive of me during my PhD studies.

I would like to acknowledge the funding support made by the National Science Foundation (NSF) Awards #1617481, #1846230, and #1953932.

VITA

Seyed Mohammadjavad Seyed Talebi

EDUCATION

Doctor of Philosophy in Computer Science **2022**
University of California, Irvine *Irvine, CA*

Master of Science in Computer Science **2019**
University of California, Irvine *Irvine, CA*

Master of Science in Electrical Engineering-Digital Electronics **2016**
Sharif University of Technology *Tehran, Iran*

Bachelor of Science in Electrical Engineering-Digital Systems **2014**
Sharif University of Technology *Tehran, Iran*

RESEARCH EXPERIENCE

Graduate Research Assistant **2016–2022**
University of California, Irvine *Irvine, California*

TEACHING EXPERIENCE

Teaching Assistant **2016**
University of California, Irvine *Irvine, California*

Teaching Assistant **2015**
Sharif University of Technology *Tehran, Iran*

REFEREED JOURNAL PUBLICATIONS

Thorough approach toward cylindrical MMW image reconstruction using sparse antenna array **2018**
IET Image Processing

Improved Two-Dimensional Millimeter-Wave Imaging for Concealed Weapon Detection Through Partial Fourier Sampling **2015**
Journal of Infrared, Millimeter, and Terahertz Waves (JIMTW)

REFEREED CONFERENCE PUBLICATIONS

Undo Workarounds for Kernel Bugs **August 2021**
USENIX Security

MegaMind: A Platform for Security & Privacy Extensions forVoice Assistants **June-July 2021**
MobiSys

Charm: Facilitating Dynamic Analysis of Device Drivers of Mobile Systems **August 2018**
USENIX Security

SOFTWARE

Undo Workaround for kernel bugs `https://trusslab.github.io/hecaton/`
USENIX'21

MegaMind `https://trusslab.github.io/megamind/`
MobiSys'21

Charm `https://trusslab.github.io/charm/`
USENIX'18

ABSTRACT OF THE DISSERTATION

Securing Mobile Devices Through Discovery, Mitigation, and Prevention of Vulnerabilities

By

Seyed Mohammadjavad Seyed Talebi

Doctor of Philosophy in Computer Science

University of California, Irvine, 2022

Professor Ardalan Amiri Sani, Chair

Mobile devices, such as smartphones and tablets, have a critical role in our everyday life. We use our mobile devices for various personal and professional tasks. Unfortunately, numerous bugs and vulnerabilities have been found in them. As a result, there are important concerns regarding the security and privacy of these devices. Compared to conventional personal computers, mobile devices have unique features such as a different processor family (e.g. ARM), different operating systems (e.g. Android), and a diverse set of I/O devices. These features raise unique challenges in securing them.

We pursued a comprehensive approach in three directions towards addressing the challenges of building secure mobile devices. In the first direction, we proposed *Charm*, a system solution to improve software analysis and bug/vulnerability discovery in mobile devices. In the second direction, we provided mitigations for existing bugs/vulnerabilities via implementing two security extensions, *bowknots*, which provide undo workaround for kernel bugs, and *MegaMind*, which provides an extensibility platform for security and privacy of voice assistants. Finally, we introduced a *split-trust machine model*, which uses physical isolation to prevent bugs from making security-critical applications vulnerable.

Chapter 1

Introduction

Mobile devices, such as smartphones and tablets, have a critical role in our everyday life. We use them for various personal and professional tasks, from sharing our family pictures on our social media to managing our bank accounts. Unfortunately, numerous bugs and vulnerabilities has been found in the mobile devices. As a result, there are important concerns regarding their security and privacy. Compared to conventional personal computers, mobile devices have unique features such as a different processor family (e.g. ARM), different operating systems (e.g. Android), and a diverse set of I/O devices. These features raise unique challenges in securing them.

Mobile devices are much more diverse than conventional personal computers. We can find them in the forms of smartphones, tablets, voice assistants, smartwatches, intelligent portable medical devices, and so on. It is reported that there are more than a thousand Android device manufacturers and more than 24,000 distinct Android devices seen just in 2015 [1]. These devices incorporate various sensors, processors, and actuators based on their specific application needs. As we need more software to manage diverse mobile hardware, the possibility of introducing bugs and vulnerabilities increases.

In addition, unlike traditional personal computers, where users mainly use a keyboard or a mouse for interaction, more diverse interaction methods are available with mobile devices. For instance, voice assistants, such as Amazon Alexa [45], Google Assistant [57], Apple Siri [67], and Microsoft Cortana [65], let you give commands to your mobile device in natural human language. Brand-new user-device interactions result in unprecedented challenges, making traditional security countermeasures ineffective. It is important to carefully study and analyze mobile devices to address these new security and privacy concerns.

A mobile device comprises different components, from user-facing components, such as user interface and applications, to system-level components, including system services, operating system, and the hardware. We focused on improving the security of the system-level components of mobile devices for several reasons. First, they are the most privileged part of a mobile device. A bug/vulnerability in them can compromise the whole system's security or completely paralyze the system's functionality. For example, hardware side-channel vulnerabilities such as the Meltdown [173] and Spectre [160] enabled untrusted applications to gain kernel privileges. Second, system-level components are a huge part of a mobile device. Only the kernel of the operating system consists of millions of lines of code. Indeed, operating system kernels are hot targets for security attacks, too. For example, according to Google, an increasing number of attacks on mobile devices are now targeting the kernel (i.e., 44% of attacks in 2016 vs. 9% and 4% of them in 2015 and 2014, respectively) [16].

We pursued a comprehensive approach in three directions towards addressing the challenges of building secure mobile devices. In the first direction, we proposed *Charm* [214], a system solution to improve software analysis and bug/vulnerability discovery in mobile devices. In the second direction, we provided mitigations for existing bugs/vulnerabilities via implementing two security extensions, *bowknots* [215], which provide undo workaround for kernel bugs, and *MegaMind* [213], which provides an extensibility platform for security and privacy of voice assistants. Finally, we introduced a split-trust machine model, which uses physi-

cal isolation to prevent bugs from making security-critical applications vulnerable. In the following sections, we discuss each of these directions in more detail.

1.1 Vulnerability Discovery

Over the years, many static and dynamic analysis solutions have been invented for system software analysis and bug/vulnerability discovery. Static analysis refers to techniques in which a tool analyzes the source code or the compiled binary of a program without executing the program. On the other hand, in dynamic analysis, the tool analyzes the program's behaviour at runtime. Symbolic execution [206, 106, 162], and taint and pointer analyses [178] are a few examples of static analysis used in mobile systems[1]. On the other side, fuzzing is an effective dynamic analysis and automatic vulnerability discovery technique, which can be applied to the operating system kernel and device drivers of mobile systems as well.

Previously, static analysis has been extensively used on mobile devices [108, 87, 198]. However, static analysis tools suffer from important limitations. They cannot uncover all the bugs and vulnerabilities. They can only detect those for which the analyzer explicitly checks for. Therefore, they might miss some unchecked types of bugs. Moreover, static analysis solutions often suffer from significant false positive rates due to imprecision. Thus, dynamic analysis needs to be adopted to combat these challenges.

The diversity of mobile devices and the inclusion of physical I/O devices create challenges for dynamic analysis, especially for mobile operating systems. A large number of highly diverse and customized device drivers are required to power the corresponding set of distinct I/O devices. Device drivers run in the kernel of the operating system and are known to be the source of many serious vulnerabilities, such as root vulnerabilities [263]. Therefore, analyzing and patching the vulnerabilities in them is crucial. Unfortunately, performing dy-

[1] In this thesis, we use the terms "mobile device" and "mobile system" interchangeably.

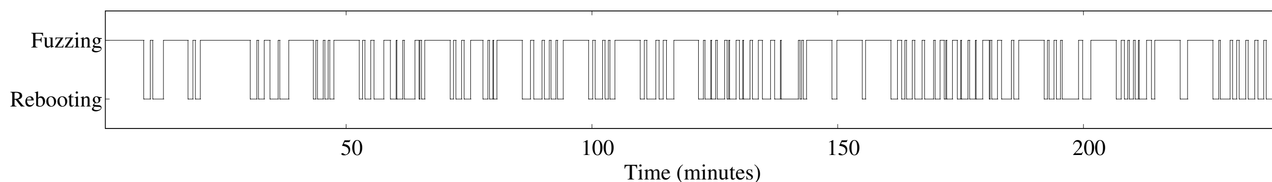


Figure 1.1: *Repetitive reboots when fuzzing the camera device driver of Nexus 5X.*

namic analysis on device drivers in mobile systems is difficult, inefficient, or even impossible, depending on the analysis. For example, a kernel fuzzer, such as kAFL [211] or Google Syzkaller [6], can be used to find various types of bugs in the operating system kernel, including device drivers. Unfortunately, fuzzing the device drivers in mobile systems encounter various disadvantages. First, using kAFL requires running the driver in an x86-based virtual machine, which is not possible for mobile drivers. Second, using Syzkaller directly on mobile systems is challenging due to (*i*) lack of support for the latest fuzzing features, such as new Syzkaller's kernel sanitizers [11, 13, 12, 14] and (*ii*) difficulty of usage due to lack of access to the system's console [7] without using a specialized hardware.

To tackle these challenges, we present Charm, a system designed to facilitate dynamic analysis of device drivers of mobile systems to find and investigate their vulnerabilities. Our key contribution in Charm that makes this possible is a systems solution for the execution of device drivers of a mobile system within a virtual machine on a different physical machine, e.g., a workstation. Such a capability overcomes the aforementioned deficiencies. That is, since the device driver executes within a virtual machine, it enables the analyst to use various dynamic analyses, including manual interactive debugging, record-and-replay, and an enhanced fuzzing. We discuss Charm's architecture and the challenges we solve implementing it in more detail in chapter 2.

Working on Charm, we identified another challenge in automatic vulnerability discovery in mobile systems. Kernel bugs, when triggered by the fuzzer, result in the reboot of the system.

Unfortunately, reboots waste a noticeable amount of fuzzing time. The reboot itself takes 10s of seconds to minutes, according to our own experience with various Android-based mobile devices and according to others [19]. In addition to wasting fuzzing time, a reboot resets the state of the system, throwing away the progress made by the fuzzer in mutating the state in order to find new bugs. Figure 1.1 shows the timeline for one fuzzing session (i.e., fuzzing the camera device driver of Nexus 5X using Syzkaller). As can be seen, reboots happen very frequently, resulting in only 44.6% of the overall fuzzer uptime being spent on fuzzing (i.e., fuzzing time). The main reason for most reboots is triggering only 6 unique bugs again and again. In chapter 3, we introduce *bowknots*, and in section 3.6.4, we show how they increase fuzzing efficiency in mobile systems by eliminating most of the unnecessary reboots.

1.2 Vulnerability Mitigation

After security analysts or automatic vulnerability discovery tools discover a bug or a design flaw in a system, they report it to the system's maintainers. Due to the complexity and diversity of current software systems, it might take a long time before the system's maintainers can fix the problem and distribute the patched software to all users. For example, bugs in several drivers of Android smartphones based on Qualcomm chipsets need to be fixed by Qualcomm. Qualcomm says: "the company hopes to patch disclosed flaws and vulnerabilities within 90 days" [20]. During this period of time, the users' systems remain vulnerable to attacks that endanger their privacy and security.

In addition, fixing some design flaws might require extreme changes to the design of the whole system. Overhauling the design of a complex system is a costly task and takes a lot of time. Hence systems' vendors might not fix those design flaws in a timely manner, and users remain vulnerable. For example, currently, voice assistants forward some of the users' requests to untrusted third-party services. Since these services can run on private

third party servers, users and the voice assistant vendor can not control what they do with users' request. Although this design flaw has been notified by security analysts, and even several security attacks proposed exploiting it [118, 185, 234, 94, 161, 28, 119], it is not yet addressed by the vendors.

Security extensions can be used to address the mentioned challenges. Security extensions extend the capabilities of a system and help users to keep their devices secure without waiting for the system's vendor to fix a bug or a design flaw in the system. Security extensions has been previously used in mobile systems to help in securing the Android operating system [142, 85]. We introduced two novel security extensions, *bowknots*, and *MegaMind*. *Bowknots* help the kernel of the Android operating system to work around existing bugs, and *MegaMind* extends the privacy and security capabilities of voice assistants. In the following sub-sections, we introduce *bowknots* and *MegaMind*, and we will discuss them in more detail in chapters 3 and 4.

1.2.1 Bowknots

Currently, the common practice to deal with kernel bugs is to find them and then patch them. There has been a lot of progress recently to automate the first step (i.e., finding bugs) [6, 211, 199, 197]. However, the second step (i.e., patching bugs) remains a highly manual and lengthy process. In practice, this requires reporting the bug to the developers of the code, e.g., the vendor in charge of a device driver, and waiting for a patch. Unfortunately, this wait can take months for the bug to sit in a queue, be evaluated by developers, and get a patch developed, tested, and merged into the kernel. While a bug is waiting for a patch, the kernel remains vulnerable posing security, reliability, and usability issues.

To solve this issue, we introduce workarounds for kernel bugs before they are correctly patched. We refer to such a workaround as a *Bug undO Workaround for KerNel sOlidiTy*

(bowknot). The key idea behind a bowknot is to undo the effects of the syscall that triggers a bug. In other words, when a syscall is issued and triggers a bug, the bowknot gets activated and neutralizes the effects of that syscall.

Undoing the syscall at arbitrary points of execution is challenging since not only a syscall can affect the kernel memory state, it can even change the state of I/O devices, e.g., a camera. The latter is especially important for device drivers, which contain most of the kernel bugs (e.g., 85% of bugs in Android kernels [237]). To address this problem, we leverage existing undo statements in error handling blocks in the kernel to generate the right undo blocks for the functions in the execution path of the bug.

A bowknot has five important properties. (*1*) it is fast to generate. (*2*) it is designed to maintain the system's functionality even if the bug is triggered. (*3*) it does not require any special hardware support. (*4*) it does not add any noticeable performance overhead. (*5*) it requires small changes to the kernel. We discuss bowknots design and implementation, and we demonstrate how bowknots can efficiently achieve the goals mentioned above in chapter 3.

1.2.2 MegaMind

Voice assistants provide a convenient user interface: natural language. However, this convenience comes with serious security and privacy risks. A voice assistant uses an always-on microphone and operates by capturing audio and sending it to the manufacturer's cloud service for processing. The cloud service transcribes the audio and interprets it as user requests. Audio recordings can have private and sensitive content, such as medical or sexual information [140]. Moreover, interpreted requests may result in unintended or unapproved actions, such as a purchase or a phone call. These unintended actions can be either due to "mistakes" by the assistant, or attacks [32, 31]. Moreover, the assistants' responses might

contain inappropriate content, such as content not suitable for children [21].

To make matters worse, voice assistants incorporate many third-party applications, which enhance the assistant functionality [69]. Unlike android or iOS applications, voice assistant's applications do not run on the voice assistant hardware. Instead, they are cloud services invoked by the manufacturer's cloud service. Researchers have shown a plethora of additional security and privacy concerns surrounding voice assistant's third-party applications [118, 185, 234, 94], including malicious skills (applications for Amazon Alexa voice assistant) [161] and unintended voice data leaks [28, 119].

To tackle this problem, we present *MegaMind*, a security and privacy extensibility platform for voice assistants. MegaMind extensions execute locally on the assistant itself. They intercept the recorded audio before sending it to the manufacturer's cloud service, and the response audio before delivering it to the user. Extensions can thus inspect, modify, or discard unwanted content to meet a user's security and privacy goals. For example, a redaction extension removes any mentions of a user's personal information from the recorded audio.

We discuss how MegaMind can efficiently improve the security and privacy of voice assistants' users in chapter 4.

1.3 Vulnerability Prevention

Despite all our efforts to discover and fix system's vulnerabilities as quickly as possible, there are still zero-day vulnerabilities that endanger computer systems and specifically the security of mobile systems. In the previous section, we discuss how to alleviate some of the damages of existing vulnerabilities through security extensions. However, the better solution is to design the system in a way that is more resilient against system's bugs and hence is more secure by design. Towards this direction, we introduced the *split-trust machine model*.

The split-trust model leverages *statically-partitioning* and *physical-isolation* to minimize the number and the complexity of hardware and software components that need to be trusted to ensure the security of a program's execution. This way, using the split-trust model, we can execute security-critical and non-critical programs side-by-side in a system.

Split-trust machine model comprises multiple trust domains, one or multiple for trusted execution environments (TEEs), one for each I/O device, one for a resource manager, and one for hosting a commodity OS and its programs. The trust domains are *statically-partitioned* and *physically-isolated*: they each have their own processor and memory (and one I/O device in the case of an I/O domain) and do not share any underlying hardware components; they can only communicate by message passing over a hardware mailbox. Moreover, we introduce a few simple, *formally-verified* hardware components that enable a program to gain provably exclusive access to one or multiple domains.

In chapter 5, we discuss our implementation of the split-trust machine model on top of a CPU-FPGA board and will demonstrate how this machine model improves the security of the system with a small added hardware cost.

Chapter 2

Charm: Facilitating Dynamic Analysis of Device Drivers of Mobile Systems

Today, mobile systems, such as smartphones and tablets, incorporate a diverse set of I/O devices, e.g., camera, display, sensors, accelerators such as GPU, and various network devices. These I/O devices are the main driving force for product differentiation in a competitive market. It is reported that there are more than a thousand Android device manufacturers and more than 24,000 distinct Android devices seen just in 2015 [1]. Therefore, one smartphone vendor might use a powerful camera so that its smartphone would stand out in this market, while another might be the first to incorporate a fingerprint scanner.

Such diversity has an important implication for the operating system of mobile systems: *a large number of highly diverse and customized devices drivers are required to power the corresponding set of distinct I/O devices.* Device drivers run in the kernel of the operating system and are known to be the source of many serious vulnerabilities such as root vulnerabilities [263]. Therefore, security analysts invest significant effort to find, analyze, and patch the vulnerabilities in them. Unfortunately, they face important deficiencies in doing

so. More specifically, performing dynamic analysis on device drivers in mobile systems is difficult, inefficient, or even impossible depending on the analysis. For example, some dynamic analyses, including introspecting the driver and kernel state with a debugger (such as GDB) and record-and-replay, requires the driver to run within a controlled environment, e.g., a virtual machine. Unfortunately, doing so for device drivers running in the kernel of mobile systems is impossible. As another example, a kernel fuzzer, such as kAFL [211] or Google Syzkaller [6], can be used to find various types of bugs in the operating system kernel including device drivers. Unfortunately, fuzzing the device drivers in mobile systems encounter various disadvantages. First, using kAFL requires running the driver in an x86-based virtual machine, which is not possible for mobile drivers. Second, using Syzkaller directly on mobile systems is challenging due to (*i*) lack of support for latest fuzzing features, such as new Syzkaller's kernel sanitizers [11, 13, 12, 14] and (*ii*) difficulty of usage due to lack of access to the system's console [7] without using a specialized hardware.

In this chapter, we present Charm, a system designed to facilitate dynamic analysis of device drivers of mobile systems in order to find and investigate the vulnerabilities in them. Our key contribution in Charm that makes this possible is a systems solution for the execution of device drivers of a mobile system within a virtual machine on a different physical machine, e.g., a workstation. Such a capability overcomes the aforementioned deficiencies. That is, since the device driver executes within a virtual machine, it enables the analyst to use various dynamic analyses including manual interactive debugging, record-and-replay, and an enhanced fuzzing.

Executing a mobile system's device driver within a workstation virtual machine is normally impossible since the driver requires access to the exact hardware of the I/O device in the mobile system. We solve this problem using a technique called *remote device driver execution*. With this technique, the device driver's attempts to interact with its I/O device are intercepted in the virtual machine by the hypervisor and routed to the actual mobile system over

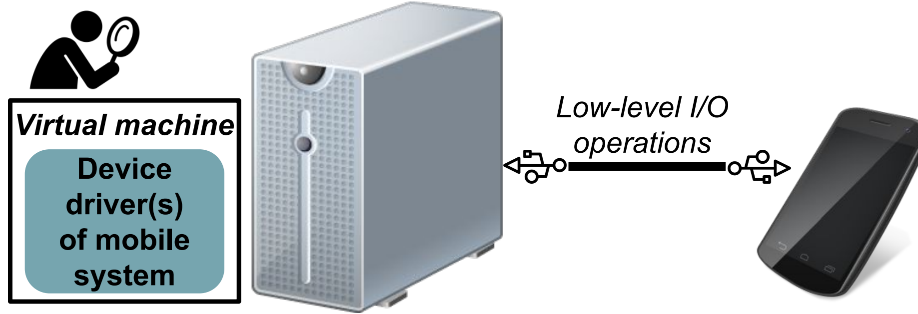


Figure 2.1: Charm enables a security analyst to run a mobile I/O device driver in a virtual machine and apply dynamic analysis to it.

a customized low-latency USB channel. In this technique, while the actual mobile system is needed for the execution of the infrequent low-level I/O operations, the device driver runs fully within a virtual machine and hence can be analyzed. Figure 2.1 shows the high-level idea behind Charm.

Remote device driver execution raises two important challenges, which we address in this work. First, interactions of a device driver with its corresponding I/O device is time-sensitive. Hence the added latency of communications between the workstation and mobile system can easily result in various time-out problems in the I/O device or driver, as our own experience with our earlier Charm prototypes demonstrated. We address this challenge with a customized USB channel. Quite importantly, *our solution does not require any customized hardware* for connection to the mobile device. It leverages the commonly available USB interface and hence will make our solution immediately available to security analysts.

Second, in addition to interacting with the I/O device's hardware, a device driver interacts with several other modules in the operating system kernel including a bus driver, power management module, and clock management module. These modules, which we refer to as "resident modules", cannot be moved to the virtual machine since they are needed in the mobile system for the usage of the USB channel. We address this challenge with a Remote Procedure Call (RPC) interface for the remote driver to interact with these modules in the mobile system. We build our RPC solution at the boundary of common Linux APIs.

Therefore, different device drivers of different mobile systems can use the same RPC interface, minimizing the engineering effort to apply Charm to new device drivers.

We implement Charm's prototype using an Intel Xeon-based workstation and three smartphones: LG Nexus 5X, Huawei Nexus 6P, and Samsung Galaxy S7. We implement remote device driver execution for two device drivers in Nexus 5X, namely the camera and audio drivers, for the GPU device driver in Nexus 6P, and for IMU sensor driver in Samsung Galaxy S7. Altogether, these drivers encompass 129,000 LoC. This demonstrate the ability of Charm to support a large range of device drivers in various mobile systems. We released the source code of Charm as well as the kernel images configured for the supported drivers. The former enables security analysts to support new device drivers, while the latter enable them to immediately apply different dynamic analysis techniques to a large set of device drivers that Charm already supports.

Using extensive evaluation, we demonstrate the following. First, we show that supporting a new device driver in Charm does not require significant engineering effort. Second, we show that despite the overhead of remote device driver execution, Charm's performance is on par with actual mobile systems. More specifically, we show that a fuzzer can execute about the same number of fuzzing programs in Charm and hence achieve similar code coverage in the driver. Third, we show that Charm enables us to find 15 bugs in drivers including one previously unknown bug (which we have reported) and one bug detected by a kernel sanitizer not available on the corresponding mobile system's kernel. Fourth, we show that we can record and replay the execution of the device driver, which, among others, can help easily recreate a bug without needing the mobile system's hardware. Finally, we show that it is feasible to use a debugger, i.e., GDB, to analyze various vulnerabilities in these drivers. Using this ability, we have analyzed three reported vulnerabilities and managed to build a arbitrary-code-execution kernel exploit using one of them.

2.1 Motivation

Our efforts to build Charm is motivated by our previous struggles to analyze the device drivers of mobile systems in order to find and understand vulnerabilities in them. In this section, we discuss three important dynamic analysis techniques: manual interactive debugging, record-and-replay, and fuzzing. We discuss the current challenges in applying them to device drivers of mobile systems and briefly mention how Charm overcomes these challenges.

2.1.1 Manual Interactive Debugging

Often security analysts use a debugger, such as the infamous GDB, to analyze a vulnerability or a reported exploit. A debugger enables the analyst to put breakpoints in the code, investigate the content of memory when and where needed, and put watchpoints on important data structures to detect attempts to modify them. Unfortunately, performing these debugging actions on device drivers are impossible as they run in the kernel of the mobile system's operating system.

Charm solves this problem. It enables the security analysts to analyze the device driver since the driver runs within a virtual machine. To demonstrate this point, we have used GDB to analyze 3 vulnerabilities in Nexus 5X camera driver (reported on Android Security Bulletins [2]). Moreover, we have also used GDB to help construct an exploit that can gain arbitrary code execution in the kernel using one of these vulnerabilities.

2.1.2 Record-and-Replay

Record-and-replay is an invaluable tool for analyzing the behavior of a program including device drivers. It enables an analyst to record the execution of the device driver and replay

it when needed. Imagine that a certain run of a device driver results in a crash (e.g., when being fuzzed). Recreating the crash might not be trivial since it might depend on a race condition that is triggered in certain interleaving of driver execution and incoming interrupts from the I/O device. However, if the execution is recorded, it can be simply replayed and analyzed (e.g., with GDB). What is extremely useful about this technique is that *the replay of the driver does not even require having access to the actual mobile system.* Therefore, anyone with access to a virtual machine can replay the device driver execution and analyze it.

While any virtual machine record-and-replay can be used in Charm, we have implemented our own solution. It records all the interactions of the driver with the remote I/O device in the hypervisor and then replay them when needed.

2.1.3 Fuzzing

Fuzzing is a dynamic analysis technique that attempts to find bugs in a software module under test by providing various inputs to the module. In case of device drivers, the input to the driver is through system calls, such as `ioctl` and `read` system calls. While fuzzing is an effective technique to find bugs in software, it often suffers from low code coverage when inputs are randomly selected. Therefore, to increase coverage, feedback-guided fuzzing techniques collect execution information and use that to guide the input generation process. One such fuzzing tool is kAFL [211], which uses the hypervisor to collect execution information of the virtual machine by leveraging the Intel Processor Tracer (PT) hardware. Using kAFL to fuzz the device drivers of mobile systems is currently impossible. However, by running the driver in a virtual machine in an x86 machine, Charm enables the use of kAFL.

Another such fuzzing tool, which is capable of fuzzing kernel-based device drivers, is Syzkaller [6], recently released by Google. Syzkaller uses a compiler-based coverage information collector,

i.e., KCOV [4], and use that to guide its input generation. Since the coverage information collector is inserted into the kernel using the compiler, it is possible to use Syzkaller to directly fuzz the device driver running inside a mobile system. Yet, using Syzkaller with Charm provides three important advantages. First, Syzkaller can benefit from other dynamic analysis techniques only available for virtual machines. Specifically, record-and-replay can facilitate the analysis of the bugs triggered by Syzkaller, as discussed earlier.

Second, it is easier to leverage new kernel sanitizers of Syzkaller in a virtual machine compared to a mobile system. Kernel sanitizers instrument the kernel at compile time to allow Syzkaller to find non-crash bugs by monitoring the execution of the kernel. Examples are KASAN [11], which finds use-after-free and out-of-bounds memory bugs, KTSAN [13], which detects data races, KMSAN [12], which detects the uses of uninitialized memory, and KUBSAN [14], which detects undefined behavior. Unfortunately, these sanitizers are not often supported in the kernel of mobile systems. To the best of our knowledge, only the Google Pixel smartphone's kernel supports KASAN [23]. In contrast, in Charm, one can simply choose a virtual machine kernel with support for these sanitizers. For example, we show that we can easily use KASAN in Charm by simply porting our drivers to a KASAN-enabled virtual machine kernel.

Finally, Syzkaller can more effectively capture and analyze crash bugs when fuzzing a virtual machine compared to a mobile system. Syzkaller reads the kernel logs of the operating system through its "console". It needs the kernel logs at the moment of the crash to capture the dump stack. The console of the virtual machine is reliably available by the hypervisor at the time of a crash. On the other hand, getting the console messages from a mobile system at the time of the crash is more challenging and requires extra hardware setup [7], which is not available to all analysts and is not easy to use. Indeed, kernel developers are familiar with the difficulty of having to use a serial cable on a desktop or laptop to get the last-second console messages from a crashing kernel in order to be able to debug the

crash. Getting the console logs from a crashing mobile system is as challenging, if not more. When such debugging hardware is not available, one can try to read the kernel messages through the Android Debug Bridge (ADB) interface, the main interface used over USB for communication to mobile systems. Unfortunately, the interface cannot deliver the kernel crash logs since the ADB daemon on the phone crashes as well. One can attempt to read the crash logs after the mobile system reboots, but crash logs are not always available after reboot since a crash might corrupt the kernel, hindering its ability to flush the console to storage. These challenges are also confirmed by the Syzkaller's developers: "Android Serial Cable or Suzy-Q device to capture console output is preferable but optional. syzkaller can work with normal USB cable as well, but that can be somewhat unreliable and turn lots of crashes into lost connection to test machine crashes with no additional info" [7]. Running the device driver in a virtual machine significantly alleviates this problem.

In our prototype, we use Syzkaller as one of the analysis tools used on top of Charm. We choose Syzkaller in order to be able to compare its performance with that of fuzzing directly on mobile systems. However, note that Charm can also support a fuzzer such as kAFL, which is impossible to use directly on a mobile system.

2.2 Remote Device Driver Execution

The key enabling technique in Charm is the remote execution of mobile I/O device drivers. In this technique, we run the device driver in a virtual machine in the workstation. We then intercept the low-level interactions of the driver with the hardware interface of the I/O device and route them to the actual mobile system through a USB channel. Similarly, interrupts from the I/O device in the mobile system are routed to the device driver in the virtual machine. Figure 2.2 illustrates this technique. We will next elaborate on the solution's details.

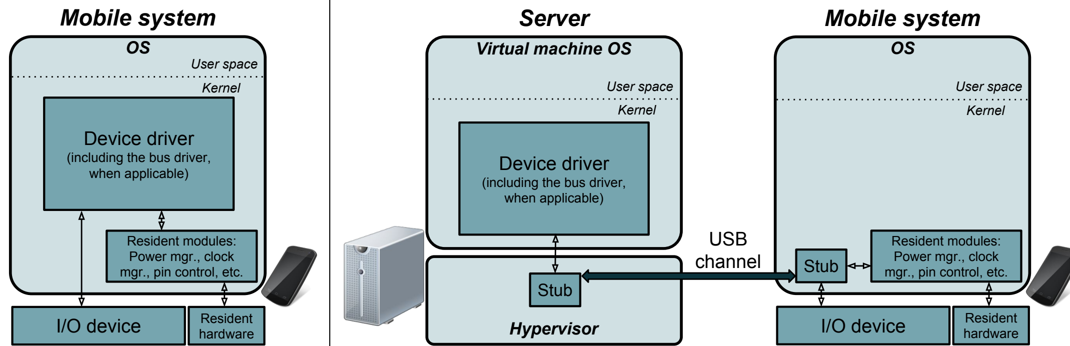


Figure 2.2: (a) Device driver execution in a mobile system. (b) Remote device driver execution in Charm.

2.2.1 Device and Device Driver Interactions

The remote device driver technique requires us to execute the device driver in a different physical machine from the one hosting the I/O device. At first glance, this sounds like an impossible task. The device driver interacts very closely with the underlying hardware in the mobile system. Therefore, this raises the question: *is remote execution of a device driver even possible?* We answer this question positively in this chapter. To achieve this, a stub module in the workstation's hypervisor intercepts and forwards the device driver's interactions with its hardware to a stub module in the mobile system, which then executes them. These interactions are three-fold: accesses to the registers of the I/O device, interrupts, and Direct Memory Access (DMA). Charm currently supports the first two. We will demonstrate that these two are enough to port and execute many device drivers remotely.

Register accesses. Using the hypervisor in the workstation, we intercept the accesses of the device driver to its registers. Upon a register write, we forward the value to be written to the stub in the mobile system. Upon a register read, we send a read request to the stub module, receive the response, and return it to the device driver in the virtual machine.

Interrupts. The stub module in the mobile system registers an interrupt handler on behalf of the remote driver. Whenever the corresponding I/O device in the mobile system triggers

an interrupt, the mobile stub forwards the interrupt to the stub in the workstation, which then injects in into the virtual machine for the device driver.

2.2.2 Device Driver Initialization

For the device driver to get initialized in the kernel of the virtual machine, the kernel must detect the corresponding I/O device in the system. Therefore, for a remote device driver to get initialized in the virtual machine, we must enable the kernel of the virtual machine to "detect" the corresponding I/O device as being connected to the virtual machine. ARM and x86 machines use different approach for I/O device detection. In an ARM machine, a *device tree* is used, which is a software manifest containing the list of hardware components in the system. In these machine, the kernel parses the device tree at boot time and initializes the corresponding device drivers. In an x86 machine, hardware detection is mainly used through the Advanced Configuration and Power Interface (ACPI). In an x86 virtual machine, the ACPI interface is emulated by the hypervisor.

The first solution that we considered was to add a remote I/O device to the hypervisor's ACPI emulation layer so that the virtual machine kernel can detect it. However, this solution would require significant engineering effort to translate device tree entries into ACPI devices. Therefore, we take a different approach. We get the x86 kernel to parse and use device trees as well. That is, we first allow the kernel to finish its ACPI-based device detection. After that, the kernel parses the device tree to detect remote I/O devices. This significantly reduce the engineering effort. To support the initialization of a new device driver, we only need to copy the mobile systems' device tree entries corresponding to the I/O device of interest into the device tree of the virtual machine.

2.2.3 Low-Latency USB Channel

We use USB for connecting the mobile system to the workstation as USB is the most commonly used connection used for mobile systems. USB provides adequate bandwidth for our use cases. For example, the USB 3.0 standard (used in modern mobile systems) can handle up to 5 Gbps.

However, in Charm, the latency of the channel between the workstation and the mobile system is of utmost importance. A high latency can result in time-out problems in both the I/O device and the device driver. In our initial prototypes of Charm, we experienced various time-out problems in the device driver and I/O device due to high latency of our initial channel implementation. In this prototype, we used a TCP-based socket over the ADB interface. However, our measurements showed that this connection introduces a large delay (about one to two milliseconds for a round trip). This latency was due to several user space and kernel crossings both in the virtual machine and mobile system. To address this problem, we implement a low-level and customized USB channel for Charm. In this channel, we create a USB gadget interface [17] for Charm and attach five endpoints to this interface. Two endpoints are used for bidirectional communication for register accesses. Two endpoints are used for bidirectional communication for the RPC calls (explained in §2.2.4). And the last endpoint is used for unidirectional communication for interrupts (from the mobile system to the workstation). In the mobile system, our stub module reads and writes to these endpoints directly in the kernel hence avoiding costly user/kernel crossings. In our KVM-based stub in the workstation, we also read and write to these endpoints directly in the kernel. Therefore, this channel eliminates all user space and kernel crossings, significantly reducing the latency.

2.2.4 Dependencies

A device driver does not merely interact with the I/O device hardware interface. It often interacts with other kernel modules in the mobile system. We use two solutions for resolving these dependencies. First, if a kernel module is not needed on the mobile system itself, we move that module to the workstation virtual machine as well. The more modules that are moved to the virtual machine, the better we can analyze the device driver behavior. An example of a dependent module that we move the virtual machine is the bus driver. Many I/O devices are connected to the main system bus in the System-on-a-Chip (SoC) via a peripheral bus. In this case, the device driver does not directly interact with its own I/O device. Instead, it uses the bus driver API.

Second, if a module is needed on the mobile system, we keep the module in the mobile system and implement a Remote Procedure Call (RPC) interface for the driver in the virtual machine to communicate with it. We have identified the minimal set of kernel modules that cannot be moved to the virtual machine. We refer to these modules as "resident modules". These modules (which include power and clock management system, pin controller hardware, and GPIO) are in charge of hardware components that are needed to boot the mobile system and configure the USB interface. We refer to these hardware components as "resident hardware". Figure 2.2b illustrates this.

Note that we implement Charm's RPC interface at the boundary of generic kernel APIs. More specifically, it uses the generic kernel power management, clock management, pin controller, and GPIO API for RPC. This allows for the portability of the RPC interface. That is, since the kernel of all Android-based mobile systems leverage mostly the same API, Charm's RPC implementation can be simply ported, requiring minimal engineering effort.

2.2.5 Porting a Device Driver to Charm

Supporting a new driver in Charm requires *porting the driver to Charm.* At its core, this is similar to porting a driver from one Linux kernel to another, e.g., porting a driver to a different Linux kernel version or to the kernel used in a different platform. Device driver developers are familiar with this task. Therefore, we believe that porting a driver to Charm will be a routine task for driver developers. Moreover, we show, through our evaluation, that non-driver developers should also be able to perform the port as long as they have some knowledge about kernel programming, which we believe is a requirement for security analysts working on kernel vulnerabilities.

Porting a device driver to run in Charm requires the following steps. The first step is to add the device driver to the kernel of the virtual machine in Charm. This requires copying the device driver source files to the kernel source tree and compiling them. Moreover, if the device driver has movable dependencies, e.g., a bus driver, the dependent modules must be similarly moved to the virtual machine kernel. One might face two challenges here. The first challenge is that the virtual machine kernel might have different core Linux API compared to the kernel of the mobile system. To solve this challenge, it is best to use a virtual machine kernel as close in version to the kernel of the mobile system as possible. This might not fully solve the incompatibilities. Hence, for the leftover issues, small changes to the driver might be needed. We have faced very few such cases in practice. For example, when porting the Nexus 6P GPU driver, we noticed that the Linux memory shrinker API in the virtual machine kernel is slightly different than that of the smartphone. We fixed this by mainly modifying one function implementation. The second challenge is potential incompatibilities due to the virtual machine kernel being compiled for x86 rather than ARM. This is due to the potential use of architecture-specific constants and API in the driver. To solve these, it is best to support the ARM constants and API in the x86-specific part of the Linux kernel instead of modifying the driver. We have faced a couple of such cases. For example, Linux

x86 support does not provide the `kmap_atomic_flush_unused()` API, which is supported in ARM and hence used in some drivers. Therefore, this function needs to be added and implemented.

The second step is to configure the driver to run in the virtual machine given that the actual I/O device hardware is not present. To do this, the device tree entries corresponding to the I/O device hardware must be moved from the mobile system's device tree to that of the virtual machine (as discussed in §2.2.2). In doing so, dependent device tree entries, such as the bus entry, must be moved too.

The third step is to configure Charm to remote the I/O operations of the driver to the corresponding mobile system. This includes determining the physical addresses of register pages of the corresponding I/O device (easily determined using the device tree of the mobile system) as well as setting up the required RPC interfaces for interactions with modules in the mobile system. The latter can be time-consuming. Fortunately, it is a one-time effort since the RPC interface is built on top of generic Linux API shared across all Linux-based mobile systems (as mentioned in §2.2.4). Hence, many of the RPC interfaces can simply be reused.

The last step is to configure the mobile system to handle the remoted operations. This needs to be done in two sub-steps. First, the Charm's stub needs to be ported to the kernel of the mobile system. This step is trivial and requires adding a kernel module and configuring the USB interface to work with the module. Second, the device drivers that are ported to the virtual machine must be disabled in the mobile system (since we cannot have two device drivers managing the same I/O device). This is easily done by disabling the device driver in the kernel build process. Alternatively, one can remove the corresponding device tree entries of the I/O device from the mobile system's device tree.

Mobile System	I/O Device	Device driver LoC
LG Nexus 5X	Camera	65,000
LG Nexus 5X	Audio	30,000
Huawei Nexus 6P	GPU	31,000
Samsung Galaxy S7	IMU Sensors (accelerometer, compass, gyroscope)	3,000

Table 2.1: Device drivers currently supported in Charm.

2.3 Implementation & Prototype

We have ported 4 device drivers to Charm: the camera and audio device drivers of LG Nexus 5X, the GPU device driver of Huawei Nexus 6P, and the IMU sensor driver of Samsung Galaxy S7. Table 2.1 provides more details about these drivers. It shows that these drivers, altogether, constitute 129,000 LoC.

As mentioned in §2.2.1, we do not currently support DMA operations. DMA is often used for data movement between CPU and I/O devices. Therefore, the lack of DMA support does not mostly affect the behavior of the driver; it only affect the data of I/O device (e.g., a captured camera frame). However, this is not always the case, and DMA can be used for programming the I/O device as well. One device driver that does so is the GPU driver. It uses DMA to program the GPU's command streamer with commands to execute. We cannot currently support this part of the GPU driver, and we hence disable the programming of the command streamer in the driver. Regardless, we show in §2.4.2 and §2.4.4 that we can still effectively fuzz the device driver and even find crash bugs.

We use a workstation in our prototype consisting of two 18-core Xeon E5-2697 V4 processors (on a dual-socket SeaMicro MBD-X10DRG-Q-B motherboard) with 132 GB of memory and 4 TB of hard disk space. We install and use Ubuntu 16.04.3 in the workstation with Linux kernel version 4.10.0-28. To support remoting of I/O operations, we have modified QEMU/KVM hypervisor (QEMU in Android emulator 2.4 used in our prototype). Note that while we use a Xeon-based machine in our prototype, we believe that a desktop/laptop-

grade processor can be used as well, although we have not yet tested such a setup. This is because, as we will show in §2.4.2, the virtual machine does not need a lot of resources to achieve good performance for the device driver. A virtual machine with 6 cores and 2 GB of memory is adequate.

We write device driver templates for Syzkaller. A template provides domain knowledge for the fuzzer about the structure of the system calls supported by the driver. Our experience with Syzkaller is that without the templates, the fuzzer is not able to reach deep code within the driver. We use these templates for all our experiments with Syzkaller in §4.8. Alternatively, one can use an automated tool for template generation, such as DIFUZE [111].

We faced a challenge for supporting interrupts. That is, the x86-based interrupt controllers supported in the virtual machine only supports up to 24 interrupt line numbers. The ARM interrupt controller in ARM, on the other hand, supports interrupts line numbers as large as 987. Hence, we extended the number of supported interrupt line numbers in our virtual machine to 128 and implemented an interrupt line number translation in the hypervisor.

2.4 Evaluation

We answer the following questions in this section: (*i*) How much engineering effort is needed to support a new device driver and a new mobile system in Charm? (*ii*) Does remote device driver execution affect the performance of the device driver? (*iii*) Is Charm's record-and-replay effective? (*iv*) Can Charm be effectively used for finding bugs in device drivers?, and Does using an x86 machine (vs. ARM) result in false positives?

2.4.1 Engineering Effort

It is important that Charm enables security analysts to easily port various drivers for analysis. We evaluate how long it takes one to port a new driver to Charm. To do this, we report the time it took my advisor to port the GPU driver of Nexus 6P and the IMU sensor driver of Samsung Galaxy S7. He ported these drivers to Charm after the implementation of Charm was almost complete, hence he could mainly focus on the port itself.

The port of these two drivers was mainly performed by a different person than me who ported the first two drivers (i.e., camera and audio drivers of Nexus 5X). Therefore, he had to learn about the port process in addition to performing the port. These two new drivers are each on a different smartphone compared to Nexus 5X used for camera and audio drivers. Therefore, the port of these drivers required adding Charm's component to these smartphone's kernel as well.

It took my advisor less than one week to port the GPU driver and, after that, about 2 days to port the sensor driver. It is worth mentioning that my advisor is familiar with kernel programming and device drivers. We believe that this is the profile of a security analyst who works on device drivers.

2.4.2 Performance

Remote I/O operations add noticeable latency to every I/O operation (i.e., register accesses and interactions with the resident modules as discussed in §2.2.4). Therefore, one might wonder if Charm will impact the performance of the device driver significantly.

To evaluate the performance of the device driver, we perform two experiments. In the first experiment, we use the Syzkaller fuzzing framework. That is, we configure Syzkaller to fuzz the driver by issuing a large number of syscalls to the camera driver of Nexus 5X both

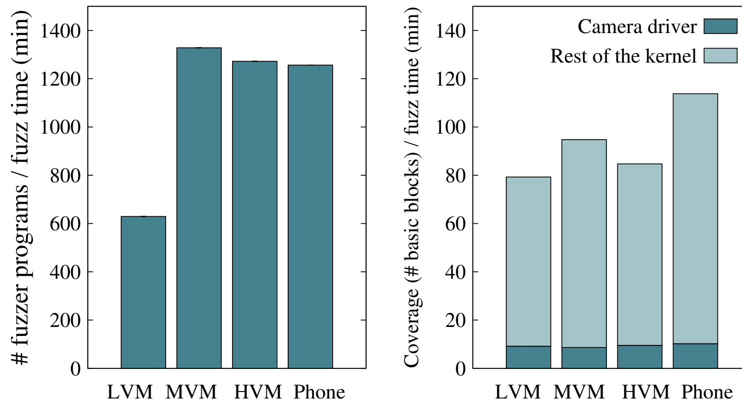


Figure 2.3: (a) Execution speed of the fuzzer. (b) Coverage of the fuzzer.

directly in the mobile system and in Charm. Syzkaller operates by creating "programs", which are ensembles of a set of syscalls for the driver, and then executing these programs. We run the Syzkaller for one hour in each experiment and measure the number of executed programs as well as the code coverage.

Figure 2.3a shows the results for the number of executed fuzzer programs per minute. We show the results for 4 setups: *LVM, MVM, HVM, and Phone*. The first three setups (standing for Light-weight VM, Medium-weigh VM, and Heavy-weight VM) represent fuzzing the device driver in Charm while the last one represents fuzzing the device driver directly on the Nexus 5X smartphone. LVM is a virtual machine with 1 core and 1 GB of memory. MVM is a virtual machine with 6 cores and about 2 GB of memory (similar to the specs of the Nexus 5X). HVM is a virtual machine with 16 cores and 16 GB of memory. Moreover, we configure Syzkaller to launch as many fuzzer processes (one of the configuration options of the framework that controls the degree of concurrency) as the number of cores. The results show that MVM achieves the best performance amongst the virtual machine setups. It outperforms the LVM due to availability of more resources needed for execution of fuzzing programs. It also slightly outperforms the HVM. We believe that this is due to the high level concurrency in the HVM experiment, which negatively impacts the performance. Finally, the results also show that MVM and HVM slightly outperform the phone's performance.

This result is important: it shows that Charm's remote device driver execution does not negatively impact the performance of the driver and hence the driver can be used for various analysis purposes.

Figure 2.3 (b) also shows the code coverage of the fuzzing experiments. It shows the coverage for the camera device driver and the rest of the kernel. The results show that Charm achieves similar code coverage in the driver compared to direct fuzzing on the smartphone. Note that the results show that the coverage in the rest of the kernel is different in Charm and in the smartphone. This is because the kernel in these two setups are different. While they are close in version, one is for x86 and one is for ARM and hence the coverage in the rest of the kernel cannot be directly compared in these setups.

In the second experiment, we choose a benchmarks that significantly stresses Charm: the initialization of the camera driver in Nexus 5X. This initialization phase, among others, reads a large amount of data from an EEPROM chip used to store camera filters and causes many remote I/O operations (about 8800). We measure the driver's initialization time on the smartphone and in MVM to be 555 ms and 1760 ms, respectively. This shows that I/O-heavy benchmarks can slow down the performance of the driver in Charm. Yet, we do not anticipate this to be the case for many dynamic analysis tools that we target for Charm including fuzzing (as seen previously).

2.4.3 Record-and-Replay

To demonstrate the effectiveness of Charm's record-and-replay, we record the execution of a PoC (related to bug #2 discussed in §2.4.4). We are then able to successfully replay the execution of the PoC and its interactions with the device driver without requiring a mobile system. Such a replay capability is a significant help for understanding this bug.

	Device driver	Bug type	Confirmed? (How?)
1	Camera	Out-of-bound memory access in msm_actuator_parse_i2c_params (**Detected by KASAN**)	Yes (LC)
2	Camera	Unaligned reg access in msm_isp_send_hw_cmd() (**Reported to kernel developers**)	Yes (PoC)
3	Camera	NULL ptr deref. in msm_actuator_subdev_ioctl()	Yes (PoC, LC)
4	Camera	NULL ptr deref. in msm_flash_init()	Yes (PoC, LC)
5	Camera	NULL ptr deref. in msm_actuator_parse_i2c_param()	Yes (LC)
6	Camera	NULL ptr deref. in msm_vfe44_get_irq_mask()	Yes (LC)
7	Camera	NULL ptr deref. in msm_csid_irq()	Yes (LC)
8	Camera	Invalid ptr deref. in cpp_close_node()	Yes (LC)
9	Camera	NULL ptr deref. in msm_ispif_io_dump_reg()	Yes (LC)
10	Camera	NULL ptr deref. in msm_vfe44_process_halt_irq()	Yes (LC)
11	Camera	NULL ptr deref. in msm_csiphy_irq()	Yes (LC)
12	Camera	NULL ptr deref. in msm_csid_probe()	Yes (LC)
13	GPU	NULL ptr deref. in _kgsl_cmdbatch_create()	Yes (MI)
14	GPU	NULL ptr deref. in kgsl_cmdbatch_destroy()	Yes (MI)
15	GPU	kernel BUG() triggered in adreno_drawctxt_detach()	No

Table 2.2: Bugs we found in device drivers through fuzzing with Charm. MI and LC refer to confirming the bug by Manual Inspection and by checking the driver's Latest Commits, respectively.

We also evaluate the overhead of recording and the execution speed of the replay. For this purpose, we record the initialization phase of the camera device driver in Nexus 5X and successfully replay it without needing a Nexus 5X smartphone. We measure the recorded initialization and the replayed initialization to take 1843 ms and 344 ms, respectively. As mentioned in the previous section, the normal initialization of this driver in Charm takes 1760 ms. The results show that (*i*) recording does not add significant overhead to Charm's execution and (*ii*) the replay is much faster than the normal execution (indeed, the replay is even faster than the initialization time on the smartphone itself, which is 555 ms). The latter finding is important: replay accelerates the analysis, e.g., for that of a PoC.

2.4.4 Bug Finding

We investigate whether Charm can be used to effectively find bugs in device drivers. We use the Syzkaller for this purpose and fuzz the drivers supported in Charm. The key question that we would like to answer is whether using an x86 virtual machine for a mobile I/O device driver would result in a large number of false positives, which can make the fuzzing efforts

more difficult for the analyst as s/he will have to filter out these false positives manually.

Therefore, for this purpose, we fuzz slightly older versions of the driver (i.e., not the latest publicly available commit of the driver). This allows us to check the bugs detected by Syzkallers against the latest patches and confirm their validity.

We also port the camera driver to a KASAN-enabled virtual machine for fuzzing with this sanitizer. KASAN detected one out-of-bounds bug in the camera driver (bug #1 in Table 2.2). This shows an advantage of Charm. Not only it facilitates fuzzing, it enables newer features of the fuzzer that is not currently supported in the kernel of the mobile system.

Table 2.2 shows the list of 15 bugs that we have found in the camera and GPU drivers (we did not find any bugs in the other drivers). The table also shows that we confirmed the correctness of 13 of these bugs through various methods (i.e., developing a PoC, checking against the latest driver commits, and manual inspection). We are still unclear about one of the bugs. We plan to use record-and-replay of Charm in Syzkaller to further analyze it.

However, we found three bugs, for which we could not find a fix in the latest driver commits. Our analysis showed that one of these bugs was a previously unknown bug caused due to unaligned access to I/O device registers. We have managed to develop a PoC for this bug as well and reported it to kernel developers already. The developers have acknowledged our report, assigned a P2-level severity [5] to it. Our analysis also shows that the other two bugs might also be unknown bugs but we have not confirmed this yet.

We believe that these results demonstrate that Charm can be used to effectively find correct bugs in device drivers through fuzzing. However, note that false positives are possible either as a result of x86 compiler bugs or incomplete driver port. For example, as mentioned in §5.4, we have not supported the DMA functionalities of the GPU driver. This can result in false positives.

Chapter 3

Undo Workarounds for Kernel Bugs

Commodity OS kernels are monolithic, large, and hence full of bugs. Bugs in the kernel cause important problems. First, they risk the system's security as some bugs might be exploitable vulnerabilities. The kernel is a highly privileged layer in the system software stack and hence is attractive to attackers. Indeed, OS kernels are hot targets for security attacks these days. For example, according to Google, an increasing number of attacks on mobile devices are targeting the kernel (i.e., 44% of attacks in 2016 vs. 9% and 4% of them in 2015 and 2014, respectively) [16]. Second, they impact the reliability and usability of the system. Even a simple crash bug, e.g., a null pointer dereference, results in a system hang or reboot, causing usability issues for the users. Even worse, bugs can corrupt the state of the software and hardware and lead to unexpected behavior. Finally, as we will show, kernel bugs can even pose practical challenges for kernel fuzzing by inducing repetitive reboots and wasting the fuzzing time.

The common practice today is to find these bugs and patch them. There has been a lot of progress recently to automate the first step (i.e., finding bugs). More specifically, several kernel fuzzers have been recently developed such as Syzkaller [6], kAFL [211], Digtool [199],

and MoonShine [197]. Indeed, these fuzzers have been successfully used to find bugs in the kernel [29, 40, 214]. However, the second step (i.e., patching bugs) remains a highly manual and lengthy process. In practice, this requires reporting the bug to the developers of the code, e.g., the vendor in charge of a device driver, and waiting for a patch. Unfortunately, this wait can take months for the bug to sit in a queue, be evaluated by developers, and get a patch developed, tested, and merged into the kernel. Our study of bugs found by Syzkaller [40] shows that bugs have taken on average 66 days to be patched. Moreover, at the time of the study (November 2019), there were several open bugs that were waiting for a patch for an average of 233 days. While waiting for a patch, the kernel remains vulnerable.

In this chapter, we introduce workarounds for kernel bugs before they are correctly patched. We refer to such a workaround as a *Bug undO Workaround for KerNel sOlidiTy (bowknot)*. A bowknot has five important properties. First, it is fast to generate. Unlike a proper patch for a bug that takes months to be ready, a bowknot takes at most a few hours. Second, it is designed to maintain the system's functionality even if the bug is triggered[1]. Kernel bugs almost always are triggered when unanticipated syscalls are issued, either by mistake by a faulty application or intentionally by malware. A bowknot undoes the side effects of this faulty or malicious syscall invocation, allowing the kernel to continue to correctly serve well-structured syscalls. Third, a bowknot does not require any special hardware support, e.g., power management support in a driver needed for checkpointing , and hence is applicable to a large number of bugs in various devices. Fourth, a bowknot does not add any noticeable performance overhead. This is because it does not do much as long as the bug is not triggered. Only when the bug is triggered, it is invoked to undo its side effects. Finally, a bowknot requires small changes to the kernel. It requires modifications only to the functions in the execution path that triggers the bug.

[1] In this chapter, we use the term "trigger a bug" to mean either executing buggy code or triggering a kernel sanitizer warning (or even a manual check) right before executing buggy code. See §3.3.1 for more details.

The key idea behind a bowknot is to undo the effects of the syscall that triggers a bug. In other words, when a syscall is issued and triggers a bug, the bowknot gets activated and neutralizes the effects of that syscall. Undoing the syscall at arbitrary points of execution is challenging since not only a syscall can affect the kernel memory state, it can even change the state of I/O devices, e.g., a camera. The latter is especially important for device drivers, which contain most of the kernel bugs (e.g., 85% of bugs in Android kernels [237]). To address this problem, we leverage existing undo statements in error handling blocks in the kernel to generate the right undo blocks for the functions in the execution path of the bug.

Bowknots, as described, achieve all the aforementioned properties, except for one. More specifically, generating a bowknot manually, while feasible, is challenging and time-consuming. Therefore, to satisfy this requirement, we introduce Hecaton, a static analysis tool that helps generate bowknots automatically[2]. Hecaton analyzes the whole kernel to find the relationship between state-mutating statements in the kernel and their corresponding undo statements in error handling basic blocks. It then uses this knowledge to generate the right undo block for the function containing the bug and the parent functions in the call stack. It also automatically inserts the undo blocks into the kernel. Due to the limitations discussed in §3.4.3, in some cases, Hecaton's automatically-generated bowknots need manual alterations. As a result, Hecaton provides a confidence score for each bowknot. This score helps the analyst determine whether a manual fix is required, before spending any time on testing the bowknot. Our evaluations with real bugs show the confidence score correctly predicts the completeness of the automatically generated bowknots in 90% of the cases.

We evaluate bowknots and Hecaton with 113 real bugs, CVEs, and automatically injected bugs in several kernel components including the IPC subsystem, networking stack, file system, and device drivers in different Android devices and x86 upstream Linux kernels. First, we show that bowknots are effective workarounds for bugs. More specifically, we show that

[2]Hecaton's source code is available at https://trusslab.github.io/hecaton/

bowknots can effectively mitigate 92.3% of real bugs and CVEs and 94.6% of injected bugs. Second, we show that bowknots manage to maintain the system functionality in 87.6% of these cases. Third, we show that Hecaton automatically generates complete bowknots for 64.6% of kernel bugs. For the rest, it only requires adding on average 3 statements and less than 2 hours of work by the analyst. Fourth, we evaluate the correctness of bowknots' undo capability with a manual case-by-case study on 10 randomly selected real bugs. We show that for 6 out of these 10 bugs, automatically generated bowknots completely undo the side effects of the buggy syscall. Fifth, we show the effectiveness of bowknots in improving the efficiency of kernel fuzzing by effectively eliminating repetitive reboots. Sixth, we empirically compare bowknots with a recent bug workaround solution, Talos [144]. Bowknots significantly outperform Talos for bug mitigation, for maintaining the system functionality, and for improving kernel fuzzing in the face of repetitive reboots. Finally, we also evaluate the performance overhead of bowknot on normal execution of kernel components. We show that bowknots' overhead is less than the baseline variations for TCP throughput and GPU framerate even if we instrument all their corresponding kernel functions with bowknots.

3.1 Motivation

3.1.1 Unpatched Kernel Bugs

As mentioned, kernel bugs pose security, reliability, and usability problems. Unfortunately, even when discovered, these bugs do not get patched immediately and there is a noticeable delay from when a bug is reported until when a patch is available. One reason behind this delay is that bugs can be complex and fixing them requires time and effort. To demonstrate this, we studied the bugs found by Syzbot [40], an automated fuzzing system based on Syzkaller [6]. At the time of the study (November 2019), there were 1691 bugs that were

fixed. Our analysis shows that these bugs took an average of 66 days to get fixed. Moreover, there were 503 bugs that were still open, for an average of 232 days.

Moreover, bugs in device drivers (which constitute 85% of the kernel bugs [237]) might take even longer as the bug needs to be reported to the developers of the driver. For example, bugs in several drivers of Android smartphones based on Qualcomm chipsets need to be fixed by Qualcomm. Qualcomm says, "the company hopes to patch disclosed flaws and vulnerabilities within 90 days" [20].

3.1.2 Problems with Unpatched Kernel Bugs

Security. The most important problem with unpatched kernel bugs is that they endanger the system's security. Bugs might be exploitable, allowing attackers to mount privilege escalation attacks. Given the high privileges of the kernel, a successful attack can be devastating for the victim's device.

Reliability and usability. Even if not exploitable, kernel bugs cause reliability and usability problems, e.g., due to a hang or reboot. Even worse, a bug might corrupt the state of the hardware and software, resulting in unexpected behavior.

Inefficient kernel fuzzing. A lesser-known problem of unpatched kernel bugs is that they cause practical problems for fuzzing the kernel by causing repetitive reboots [218]. Kernel bugs, when triggered by the fuzzer, result in the reboot of the system. Unfortunately, reboots waste a noticeable amount of fuzzing time. The reboot itself takes 10s of seconds to minutes according to our own experience with various Android-based mobile devices and according to others [19]. In addition to wasting fuzzing time, a reboot resets the state of the system, throwing away the progress made by the fuzzer in mutating the state in order to find new bugs.

Unfortunately, modern feedback-driven fuzzers such as Syzkaller and AFL may trigger the same bug many times resulting in *repetitive reboots*, i.e., costly and useless reboots caused by the same bug, due to the feedback-driven fuzzing algorithm [26, 18] and some bugs being easy to trigger.

Figure 1.1 shows the timeline for one of these fuzzing sessions (i.e., fuzzing the camera device driver of Nexus 5X using Syzkaller). As can be seen, reboots happen very frequently, resulting in only 44.6% of the overall fuzzer uptime being spent on fuzzing (i.e., fuzzing time). The main reason for most reboots is triggering only 6 unique bugs again and again.

3.1.3 Current Approaches

Approach I: mitigation through code disabling. One possible approach is to try to mitigate a bug by disabling the part of the code that contains the bug. This can be done at different granularities. For example, the buggy subcomponent within the code can be disabled. If applied to the kernel, one can imagine disabling a device driver if it has a bug. It can also be applied at the function level. Talos uses this approach [144]. It neutralizes a vulnerability in a codebase by disabling the function that contains it. The function instead is instrumented to return an appropriate error message.

Although disabling the code can mitigate the bugs and vulnerabilities in many cases, it very likely breaks the system functionality. Losing functionality in a system will deter the use of this approach in practice. This approach does not help with the kernel fuzzing efficiency either. This is because code disabling limits the code coverage of the fuzzer (see §3.6.1 and §3.6.4 for empirical results).

Approach II: dirty patching. One might wonder whether the analyst can perform a "quick and dirty patch" to fix the bug. For example, if the bug is a null pointer dereference,

they can add a null pointer check to return directly to avoid crashing. Unfortunately, dirty patching suffers from similar drawbacks as code disabling. That is, it can break the functionality of the system or result in unexpected behavior if not done carefully. In addition, such patches might still need engineering effort. For example, a dirty patch for a use-after-free bug resulting from a race condition is not trivial.

3.2 Overview

3.2.1 Goals

Our goal is to design a bug workaround solution that can mitigate the undesirable side effects of a bug until a proper patch is available. In other words, the applicability of the workaround is in the window of vulnerability from when the bug is first discovered until when the correct patch is available.

The main users of kernel bug workarounds are kernel security analysts, OS vendors, and IT departments. For example, the security team in an OS vendor company might find a bug and report it to the corresponding developers, e.g., another company in charge of a device driver or a development team within the same company. While they wait for the patch, they can use a workaround to mitigate the bug. Or an IT department might apply a workaround for a known bug in the company's servers or employees' workstations. Finally, security analysts can leverage this tool to mitigate kernel bugs in their own devices, e.g., to improve the efficiency of their kernel fuzzing sessions. To show our solution's applicability, we implement and test it on several targets, such as ARM-based Android smartphones and x86-based Linux kernels.

We identify five important properties that a bug workaround solution must satisfy. First,

it should be fast to generate, otherwise it will not be available soon enough to help in the aforementioned window of vulnerability. Second, a workaround for a kernel bug should maintain the system's functionality even if the bug is triggered. Third, the workaround approach should be widely applicable to different kernel components and different kernels. Moreover, it should not require special hardware support, e.g., to checkpoint the state of an I/O device. Fourth, a workaround should not add any noticeable performance overhead. Finally, a workaround should require small changes to the kernel, otherwise it will not be accepted by vendors for release in the window of vulnerability.

3.2.2 Key Idea & Design

Bowknots. In this chapter, we introduce a workaround for kernel bugs called *Bug undO Workaround for KerNel sOlidiTy (bowknot).* The key idea behind a bowknot is to undo the effects of the in-flight syscall that triggers a bug. That is, if a syscall is issued and triggers a bug, the bowknot generated for that bug undoes the syscall and returns, effectively neutralizing the syscall. It is important to note that a bowknot does not disallow a syscall, e.g., disallow all `ioctl` syscalls. It allows the syscall to be used as long as it does not trigger the bug. Only when an invocation of the syscall results in the bug getting triggered (e.g., due to using unexpected inputs), the bowknot kicks in to undo it so that the system can continue its execution and serve other well-structured syscalls.

Bowknots protect the kernel from corruption, which is critical for continued use of the system. They, however, can impact the program issuing the syscall. For example, they might result in the program breaking or terminating with an error message. We believe this is acceptable for three reasons. First, we do not anticipate most kernel bugs to be triggered by well-behaved applications. Many kernel bugs are only triggered when a meticulously-crafted syscall is issued, typically by malware. Second, applications can be restarted, if corrupted. Finally,

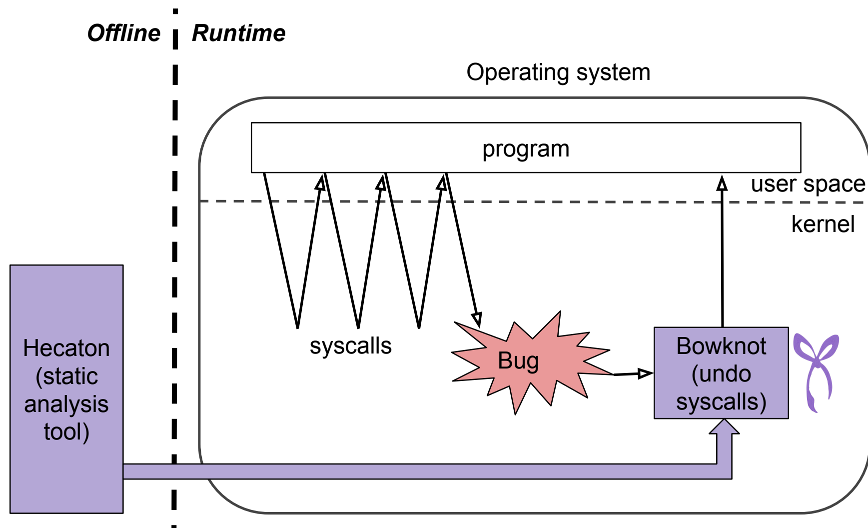


Figure 3.1: *High-level idea behind bowknots and Hecaton.*

kernel bugs that unconditionally break the usability of well-behaved applications are rare. This is because the kernel is tested for basic functionality by kernel developers.

Hecaton. Bowknots, as described so far, satisfy all but one of the aforementioned properties. More specifically, generating them manually requires noticeable engineering effort as one needs to study the execution path that triggers the bug and figure out how to undo the syscall. Therefore, to satisfy this last property, we introduce Hecaton, a static analysis tool that generates bowknots and inserts them into the kernel automatically. To do so, Hecaton leverages existing *undo statements* found within *error handling blocks* in the kernel to generate the right undo blocks for the functions in the execution path of the bug. Existing error handling blocks in the kernel undo the effects of a syscall on the software and hardware state in case of *expected* errors, such as a null pointer or a busy I/O error in some fixed code locations. While the kernel does not have error handling code for arbitrary bug sites in the execution of a syscall, the idea in Hecaton is to leverage existing undo statements in these blocks to generate the right undo code needed for a bowknot. More specifically, Hecaton leverages existing error handling blocks to discover undo statements for each state-mutating statement. Using such knowledge, Hecaton can then automatically generate the required bowknot for different functions. Figure 3.1 shows the high-level idea behind bowknots and

Hecaton.

3.2.3 Workflow

Assume that the OS analyst has identified a bug in the kernel and would like to apply a bowknot to it. They take the following steps to achieve this.

In the first step, they need to identify the functions in the execution path from the beginning of the syscall until where the bug is triggered, i.e., the call stack. The call stack must include the inline functions since it will be used by Hecaton, which operates at the source code level. Bugs found by Syzkaller, such as the reported bugs in the Syzbot system [40], come with enhanced call traces, including all the inline functions and their location in the source code. For other bugs, the analyst can use any tool to find the stack. However, finding the inline functions in the stack might not be trivial. To make this step easy for the analyst, we provide support in Hecaton. That is, Hecaton instruments all the functions in the kernel component under study with some logging messages. The analyst then executes the Proof-of-Concept (PoC) program of the bug, checks the kernel logs, and extracts the list of functions executed in the syscall. They then feed this list back into Hecaton, which uses it to generate a copy of the kernel where only these functions are instrumented with bowknots. Hecaton provides a confidence score for each bowknot. If all the confidence scores for the instrumented functions are higher than a predefined threshold, the analyst goes to the next step to test the instrumented kernel. Otherwise, they can decide to investigate the bowknots with low confidence score and manually correct them, or altogether drop working on these bowknots if they are unwilling to spend time and manual effort to fix the bowknots.

The analyst then tests the instrumented kernel using the PoC and test programs. The purpose of test programs is to demonstrate proper functionality of the system after undo by bowknots. More specifically, the analyst first runs the PoC to verify that it does not succeed,

e.g., it does not crash the kernel. They then run the tests to verify that the kernel component under test is still functional. If either fails, the analyst checks the generated bowknots. The analyst spends a few hours (e.g., up to 2 hours in our evaluation) to identify the problem, e.g., a missing undo statement. In fact, some of the bowknots might have explicit warnings from Hecaton (§3.4.2), which makes the manual step more straightforward. After a fix, they run the tests again. If the analyst does not find a fix in this period (e.g., the two hours), they declare the use of bowknots ineffective.

It is noteworthy that the analyst does not even need a fully functional PoC to test the bowknots. A program that results in the execution of the same functions but does not even trigger the bug suffices. We have indeed used this in our own evaluations. We tested a reported PoC that reached the bug but did not trigger it. Yet, by adding an explicit crash just before the bug site, we emulated the behavior and tested the undo behavior by the bowknot.

Finally, we note some bugs might be triggered through more than one call stacks. While such bugs are not common, to mitigate them, the analyst needs to generate bowknots for each call stack separately.

3.3 Bowknots

Bowknots are workarounds for kernel bugs. The key idea behind bowknots is to undo the side effects of the syscall that triggers the bug. More specifically, bowknots undo the side effects of state-mutating statements from the syscall's kernel entry point until where the bug is triggered. We define a state-mutating statement as one that alters the state of the kernel or an underlying I/O device.

For example, imagine a camera device driver `ioctl` syscall, which when called, allocates

a memory buffer using `kmalloc()`, acquires a spin lock (`spin_lock()`), and turns on the flash for the camera (using the hypothetical function `turn_on_flash()`). Now imagine there exists a bug after this where a pointer might be null depending on the syscall input. To mitigate this bug, the analyst can apply a bowknot. It first turns off the camera flash (by calling `turn_off_flash()`), unlocks the spin lock (by calling `spin_unlock()`), and frees the allocated memory buffer (by calling `kfree()`). As can be seen, the state of the system (including the kernel memory state as well as the I/O hardware state, e.g., the camera hardware state) after undo is the same as the state before issuing the syscall. Therefore, the system can now resume its execution as if the syscall did not happen.

Our strategy for undoing a syscall is to leverage existing undo statements in error handling code in the kernel to generate the proper undo code that undoes the effects of all state-mutating statements in the syscall. Existing error handling code in the kernel undoes the effect of these statements when facing an expected error. The insight behind this approach is that OS kernels have to be robust and handle various corner cases or errors. Therefore, we attempt to reuse the existing undo statements to generate the right undo code for a bug location. In this section, we show how a bowknot can be used for a bug. In the next section, we discuss how Hecaton helps to automatically generate the undo code for bowknots.

3.3.1 Function Instrumentation

The goal of function instrumentation for a bowknot is to undo the executed statements in a function when a bug is triggered. We support two types of bowknots for a function: automatically-triggered and manually-triggered. Automatically-triggered bowknots are the common ones and are used for crash bugs and bugs automatically detected by a kernel sanitizer. The manually-triggered ones are for more complex bugs, such as race conditions and memory leaks.

```
1  #define CGOTO if(unlikely(current->bowknot_flag))
2                    goto bowknot_label
3
4  long kgsl_ioctl_device_waittimestamp_ctxtid(
5       struct kgsl_device_private *dev_priv, unsigned int cmd,
6       void *data)
7  {
8    uint64_t bowknot_pairmask = 0;
9
10   struct kgsl_device_waittimestamp_ctxtid *param = data; CGOTO;
11   struct kgsl_device *device = dev_priv->device; CGOTO;
12   long result = -EINVAL; CGOTO;
13   struct kgsl_context *context; CGOTO;
14
15   mutex_lock(&device->mutex); CGOTO;
16   bowknot_set_bit(bowknot_pairmask, 2);
17
18   context =
19     kgsl_context_get_owner(dev_priv, param->context_id); CGOTO;
20   bowknot_set_bit(bowknot_pairmask, 1);
21
22   if (context == NULL) {
23      goto out;
24   }
25   ...
26 out:
27   kgsl_context_put(context); CGOTO;
28   bowknot_unset_bit(bowknot_pairmask, 1);
29   mutex_unlock(&device->mutex); CGOTO;
30   bowknot_unset_bit(bowknot_pairmask, 2);
31   return result;
32
33   if (bowknot_global_always_false < 0) {
34 bowknot_label:
35     current->bowknot_flag = 0;
36     if(bowknot_check_bit(bowknot_pairmask, 2))
37      mutex_unlock(&device->mutex);
38     if(bowknot_check_bit(bowknot_pairmask, 1))
39      kgsl_context_put(context);
40     current->bowknot_flag = 1;
41     return -1;
42   }
43 }
44
45 long kgsl_ioctl(struct file *filep,
46                unsigned int cmd, unsigned long arg)
47 {
48   ...
49   ret = kgsl_ioctl_device_waittimestamp_ctxtid(...); CGOTO;
50   ...
51   if (bowknot_global_always_false < 0) {
52 bowknot_label:
53   ...
54     return -1;
55   }
56 }
```

```
     ...
15   mutex_lock(&device->mutex); CGOTO;
16   bowknot_set_bit(bowknot_pairmask, 2);
17
   if(unlikely(param == unexpected_ctx))
      goto bowknot_label;
18   context =
19     kgsl_context_get_owner(dev_priv, param->context_id); CGOTO;
     ...
```

Figure 3.2: *Example function in the Qualcomm KGSL GPU device driver after instrumentation with a bowknot. (Up) Automatically-triggered, (Down) Manually-triggered bowknot. The blue and bold text highlights the automatically added code. The green and italic text highlights the manually added lines. The code presented here is slightly modified from the actual function code and from the one generated by Hecaton for better readability.*

Automatically-triggered bowknots. Figure 3.2 (Up) shows an instance of an automatically-triggered bowknot for a function in Qualcomm's KGSL GPU driver. This function is the handler for one of the supported `ioctl` syscall commands for this driver and is called by the main `ioctl` handler, `kgsl_ioctl`. The function instrumentation has several parts. The first part is an undo block at the end of a function, which contains all the undo statements corresponding to the state-mutating statements in the function. There are two state-mutating statements in this function: `kgsl_context_get_owner()`, which returns a `context` object while incrementing its reference counter, and `mutex_lock()`, which acquires a lock. The corresponding statements to undo the effects of these statements in the function are, respectively, `kgsl_context_put()` and `mutex_unlock()`. This undo basic block is also protected by an always-false global variable (`bowknot_global_always_false`) preventing it from being used in the normal execution of the function. It is only accessible through explicit jumps to `bowknot_label`.

The second part of the instrumentation is for detecting, at runtime, the state-mutating statements that are executed before the crash. This is because not all execution paths within a function execute the same set of state-mutating statements. If not taken into account, in the case of a specific bug, an unnecessary undo statement might get executed. Therefore, we instrument the function to keep track of the execution of the state-mutating statements. To do this, we use a per-function mask variable. We add the mask update statements after each state-mutating and undo statement. We also make the undo statements in the undo block conditional based on the bits in this mask. In our example, after a call to `mutex_lock()`, we set a bit in the mask variable. After a call to the corresponding `mutex_unlock()`, we reset the same bit in the mask variable. Then in the undo block of the bowknot, we check the bit. If set, we execute the `mutex_unlock()` statement.

The third part of the instrumentation, which is used for automatically-triggered bowknots, is the automatic redirection of the execution to the undo block when a bug is triggered. To

do this, we add conditional `goto` statements (`CGOTO`) after all statements. The goal of these statements is to redirect the execution to the undo block in case of a bug. When a crash happens or a bug is detected by the kernel sanitizer, the execution is redirected to the kernel exception handler, which we instrument. Our exception handler code sets the redirection flag (`bowknot_flag`), which is a thread-specific flag, and then returns the execution back to the function resulting in a jump to the undo block. In the previous example, assume that `param` is null and results in a crash at line 19. The exception handler is then invoked, sets the flag, and resumes the execution in the function (by skipping the crashing instruction), which then executes the conditional `goto` statement in the same line and jumps to the undo block. This condition is typically false during normal execution in the kernel. Hence, we use the compiler's `unlikely` directive, which helps with performance in normal execution by instructing the compiler to insert some instructions in the binary to assist CPU's branch prediction.

We also support automatic redirection for bugs detected by a kernel sanitizer (if activated, e.g., during a fuzzing session). In this case, we force-execute the kernel exception handler for bugs detected by the sanitizers, e.g., memory safety bugs detected by KASAN [11].

Note that automatically-triggered bowknots only get triggered on system crashes and warnings generated by kernel sanitizers. As a result, for non-crashing bugs that can potentially result in kernel corruption, the security of automatically-triggered bowknots depends on the appropriate use of kernel sanitizers (e.g., KASAN and KMSAN) to catch the bug before the corruption happens. Although currently sanitizers are enabled only during testing due to their memory and performance overhead, there are recent efforts to enable efficient sanitizers to be used in deployed products as well[224][165].

Manually-triggered bowknots. There are two important scenarios when manually-triggered bowknots are desired or needed. First, some bugs do not result in a crash nor are detected by a kernel sanitizer. However, the security analyst knows the condition under

which the bug is triggered. In this case, the analyst can add an explicit condition to the function containing the bug to redirect the execution to the undo block before the bug is triggered. Figure 3.2 (Down) shows an example. In this (hypothetical) case, if the `param` parameter is equal to a known global object, the behavior is buggy resulting in the corruption of the object. Therefore, the analyst can add the conditional block between lines 17 and 18 to jump to bowknot's undo block. The analyst does not need to generate the bowknot nor figure out which undo statements need to be called. She only needs to determine where and under what conditions the bowknot needs to be executed.

Second, in some production systems, instrumenting the kernel exception handler or deploying a kernel sanitizer (as needed for automatically-triggered bowknots) might not be acceptable. In such cases, manually-triggered bowknots can be used, even for simple bugs such as crash bugs.

3.3.2 Recursive Undo of Call Stack

When a bug is triggered, bowknot executes the undo code for the function the bug is in. It then needs to undo the effects of the statements in the parent functions.

To do this, we undo the parent functions similar to the buggy function. Figure 3.2 shows the parent function as well. We perform the recursive undo through the use of the thread-specific flag mentioned earlier (`current->bowknot_flag`). When returning from the buggy function, this flag is set. Moreover, the parent function is also instrumented with the conditional `goto` statements. Therefore, after returning from the buggy function, the parent function jumps to its own bowknot and executes its own undo code. This recursive undo continues until the syscall returns, at which point the flag is cleared.

It is important to note that the bowknots in the parent functions are always automatically-

triggered. Only the last function in the stack might need manual triggering of the bowknot.

Also, note that it is feasible to rely on the existing error handling blocks in some functions rather than using bowknots. We use this approach for the first few functions in the execution paths of a syscall, which receive a syscall and route them to an underlying component to handle. As a practical guideline, when dealing with a bug in a specific kernel component, e.g., a device driver, we only apply bowknots to the functions in the path within the driver. When recursively undoing the functions, the entry function in the kernel component simply returns an error, which is elegantly handled by existing kernel code by routing the error to the user space. We take this approach for two reasons. First, the functions parsing and routing a syscall are triggered for every syscall and hence have impact on the system's performance. Second, these functions are mature and have adequate error handling code, eliminating the need to inject custom undo code for them.

3.4 Automatic Generation of Bowknots

In this section, we describe how Hecaton generates the undo block of the bowknot automatically. Hecaton also automatically instruments the designated kernel functions, which we do not discuss further here.

We build Hecaton as a static analysis tool. It generates the undo block by analyzing the entire kernel to infer the relationship between state-mutating statements and their corresponding error handling undo statements. Hecaton achieves so in two main steps: (*i*) generating a kernel-wide knowledge database of function pairs and (*ii*) generating the undo block using the database as well as function-level analysis. We next describe these two steps.

3.4.1 Function-Pair Knowledge Database

The goal of the function-pair knowledge database is to store pairs of functions that mutate and undo the kernel state. In other words, a state-mutating function and an undo function are paired, if the latter undoes the effect of the former. (`kmalloc, kfree`), (`mutex_lock, mutex_unlock`), and (`msm_camera_power_down, msm_camera_power_up`) are a few examples of such function pairs. The function-pair knowledge database can be reused across various kernels, e.g., the kernels of different Android devices, with minimal changes. Therefore, our general approach is to automatically extract function pair candidates, manually inspect them, and add them to the database if verified. This approach provides high confidence in the database. Moreover, since generating the database is mostly a one-time effort, the manual effort is not significant. (We provide some quantification of the manual effort later in this section and in §3.6.2).

Identifying function pair candidates. Hecaton statically analyzes the entire kernel to identify function pair candidates. It uses two methods to identify the candidates. First, it uses the function names. In this method, Hecaton considers a function pair as a candidate, if the names of two functions only differ in one word and the difference is one of the following: (`put, get`), (`put, create`), (`release, get`), (`release, create`), (`remove, create`), (`deinit, init`), (`unregister, register`), (`unlock, lock`), (`down, up`), (`disable, enable`), (`sub, add`), (`dec, inc`), (`unset, set`), (`clear, set`), (`free, alloc`), (`stop, start`), (`suspend, resume`), (`disconnect, connect`), (`unmap, map`), (`dequeue, enqueue`), (`unprepare, prepare`), and (`detach, attach`). Using this method, for example, Hecaton found 540 pairs of function in the Linux kernel used in the Pixel3 smartphone.

Unfortunately, not all function pairs differ in one word only. As a result, Hecaton employs a second method, in which it uses existing error handling blocks in the kernel to identify undo functions and then match them to candidate state-mutating functions in the same

function using string matching. More specifically, Hecaton marks all the functions in error handling blocks as undo functions. Then, for each undo function, it matches it with a candidate state-mutating function in the same function using similarity in their names and input/output variables. For the similarity score, Hecaton calculates the sum of the lengths of all mutually-exclusive substrings. To do so, Hecaton finds the longest common substring (LCS) and adds its length to the similarity score. Then it deletes the LCS from both strings and repeats the previous steps recursively until there is no common substring with more than two characters.

Towards this goal, Hecaton needs to be able to identify error handling blocks in the kernel. Hecaton does so by looking for common conditional statements used to identify and handle an error in the kernel. By investigating a large amount of kernel code, we have identified four such conditional blocks including (*i*) `if (rc < 0) {...}` where `rc` is an integer, (*ii*) `if (IS_ERR(p)) {...}` or `if (p == NULL) {...}`, where `p` is a pointer, (*iii*) `if (...) {...; return ERROR;}` where `ERROR` is a constant negative integer, often one of the commonly used error numbers in the kernel such as `-ENOMEM` and `-EFAULT`, and (*iv*) `if (...) {...; goto LABEL;}`. It also considers simple variations of these four categories such as checking within the `else` block rather than the `then` block for categories (*iii*) and (*iv*).

Once it identifies the error handling blocks, Hecaton needs to match the undo functions in them with state-mutating functions. That is, it assumes that every undo function call statement undoes the effects of a single state-mutating function call in the same parent function. For example, `kfree()` is an undo function statement that corresponds to the state-mutating function statement `kmalloc()`. Hecaton uses the same heuristic string matching discussed above to identify the candidates. For example, `kgsl_context_put(context)` is paired with `context = kgsl_context_get_owner(...)`. To do this, Hecaton calculates the string-based similarity score between the undo statement and all statements prior to the corresponding error handling block. It then chooses the function with the highest similarity

score. Using this method, for example, we identified 1158 candidate pairs in the Pixel3 kernel (excluding the pairs found using the previous method).

Manual inspection of function pair candidates. Not all function pair candidates are true pairs of state-mutating and undo ones. This is because the method discussed above, i.e., string matching, is not precise. Therefore, we perform manual inspection on the candidates to identify the true pairs. In this step, we use our knowledge of kernel code. In addition, we use the frequency of appearances of a function pair candidate as a hint to facilitate the manual inspection. Pairs that appear many times together in many functions are less likely to be false pairs. Using manual inspection, in the case of the Pixel3 kernel, we verified all 540 pairs identified using the first method and 658 of the function pairs identified using the second one, bringing the total number of function pairs in the database to 1198. This manual inspection took me 7 days to complete. However, as mentioned, this is largely a one-time effort. Supporting a new version of the kernel or a new device driver adds a small number of new candidate pairs, which can be verified fast. As an example, once we had the database for the Pixel3 kernel, we ran our static analysis tool on a Nexus 5X driver that we needed to test. Doing so resulted only in 9 new candidate pairs, which we quickly inspected. We evaluate the amount of manual effort for x86 kernels in §3.6.2.

3.4.2 Generating the Undo Block

To generate the bowknot's undo block, we need to identify all the state-mutating statements in the function, and generate the corresponding block. Hecaton is not currently able to generate an undo statement, as it might require fixing the parameters passed to a function. Therefore, Hecaton tries to *reuse existing undo statements* in a function and match them with the state-mutating ones. If Hecaton does not find a match for an undo statement in a function, or if it does not find a match for a state-mutating one, it inserts a warning in

the undo blocks that it generates so that the analyst can manually fix the problem. Simply reusing existing statements is adequate in a large number of functions (§3.6.1).

As mentioned, Hecaton attempts to find all undo statements in the function for which it generates the undo block. An undo statement might be a function call or not. Hecaton uses the knowledge database to identify all the undo function call statements. For other undo statements, e.g., a counter decrement, it relies on the error handling blocks in the function.

To identify the error handling block candidates, we use the patterns often used for these blocks as discussed earlier. In addition, we also inspect all blocks that have one of the following jump statements in their bodies: `break`, `continue`, `return`, and `goto`. If such a block contains an undo function call (determined by consulting our knowledge database), we mark that block as an error handling one as well. In addition to the error handling blocks, some functions incorporate undo statements prior to the return statement. For example, it is common in kernel functions to allocate, acquire, enable, or turn on a resource, perform a task on it and then free, release, disable, or turn off that resource before returning a success value. Hecaton reuses these undo statements as well.

Having all the undo statements, the next step is to find their corresponding state-mutating statements. For error handling statements that are function calls, Hecaton uses its knowledge database. If there are multiple instances of the same state-mutating function, Hecaton chooses the one that shares more variables with the error handling statement. For all other types of statements, Hecaton uses string matching to pair them with state-mutating statements.

3.4.3 Incompleteness and Confidence Score

As mentioned, a small portion of bowknots generated automatically by Hecaton are not complete and require manual amendments. We analyze the underlying reasons for this incompleteness through experiments and a case-by-case study. We enable Hecaton to automatically detect features in functions that may result in the generation of an incomplete bowknot. For each generated bowknot, Hecaton provides a confidence score, indicating the probability of its effectiveness. Also, in cases that manual effort is necessary, Hecaton highlights the function(s) in the call stack that have the most negative effect on the confidence score and need manual corrections. Our experience and analysis show that six features play critical roles in generating complete bowknots. We quantify these features and linearly combine them into a single confidence score using adjustable coefficients. Finally, we tune these adjustable coefficients using real bugs (§3.6.1).

The first feature we use is the **location of the bug**. Our experience shows that if the last function of the call stack of the bug is inside a kernel component (e.g., a device driver), it is more likely that Hecaton could generate a complete working bowknot. In cases that the bug is in core kernel, for example, inside an inline function that manipulates kernel objects, it is less likely that Hecaton could generate complete bowknots.

The second feature is the presence of **missing undo statements**. As we discuss in §3.4.2, Hecaton currently cannot generate undo statements from scratch. We decrease the confidence score when Hecaton does not find an undo match for a state-mutating function found in its knowledge database.

The third feature is the **method of error handling block detection** used in a function. As we discuss in §3.4.2 and §3.4.1, Hecaton uses different patterns to identify error handling blocks. Some of these patterns are used both in error handling and non-error handling blocks and hence might produce false undo statements. Therefore, we decrease the confidence score

if such patterns are used.

The fourth feature is the **presence of function pointers**. As Hecaton currently cannot pair the state-mutating function pointers with its correct undo statement using its knowledge database, it solely relies on the string matching heuristic to pair them. As a result, we decrease the confidence score in the presence of such statements.

The fifth feature is the **presence of multi-statement undo code**, where multiple statements are used to undo one or more state-mutating statements. One important example is when a loop is used to undo the effects of another loop. Another important example is when a critical section is used in the error handling block. Hecaton assumes a one-to-one mapping between state-mutating and undo statements, and hence does not currently automatically handle such cases.

Finally, to take the miscellaneous unknown sources of inaccuracy in Hecaton's static analysis into account, we decrease the confidence score as the **number of state-mutating statements** in a function increases since having more state-mutating statements to pair increases the error probability.

3.5 Implementation

Static analysis tool. We implement Hecaton in C++ and Python with about 4,550 LoC. We use Clang for static analysis in Hecaton as it allows us to perform the analysis at the source code level. While we mainly test our solutions with the Linux kernel of Android devices and upstream x86 Linux kernels, we note that they are applicable to other OSes as well. Our static analysis tool is implemented as a plug-in for the Clang compiler. We use our plug-in alongside Android Clang version 5.0.1 for our Android devices, and we use the same plug-in (with a small modification to make it compatible with the newer version of

Clang) alongside Clang version 11.0.0 for our upstream x86 Linux kernels.

We perform our analysis on the Abstract Syntax Tree (AST). When using the AST, we do not need to worry about parsing and lexing the source code. Moreover, we have high-level information of the source code needed for our analysis, such as functions and variables names. In addition, the organized structure of the AST facilitates finding the error handling blocks. In AST, all the statements and expressions are organized in a hierarchical structure as nodes of a tree, and Clang provides many helper functions to traverse the AST in an efficient way. There are also many helper functions to obtain attributes of each node of the AST. To obtain the AST of the source code, we use `ASTFrontendAction` with a custom `ASTConsumer`. We override the `VisitFunctionDecl` function of our custom `ASTConsumer` to obtain all the function declaration nodes in the AST. All the statements in the body of each function appear as children nodes of the function declaration node. To perform our analysis, we recursively visit all the children nodes in several passes. In these passes, using AST, first, we identify and pair undo nodes and state-mutating nodes to generate a bowknot for each function. As discussed in §3.3.1, a bowknot includes a generic undo block, several conditional `goto` statements, and several mask update statements. Then, using the AST helper function, `getSourceRange`, we identify the locations of these nodes in the source files. Finally, using Clang's Rewriter tool, we directly inject the generated bowknot into the source code.

Exception handler. We have implemented Hecaton with automatically-triggered bowknots for two Android devices naming Pixel3 and Nexus 5X and various versions of three x86 kernel branches naming upstream Linux kernel, Google's KMSAN kernel, and Linux-Next kernel. Nexus 5X runs CyanogenMod-13 Android OS with Linux kernel 3.10.73, Pixel3 runs Android-9.0.0 r0.43 with Linux kernel 4.9.96, and the x86 Linux versions vary between 5.5.0 and 5.8.0.

As discussed in §3.3.1, to support automatically-triggered bowknots, we need to instrument

the kernel's exception handler. First, we need to distinguish between bowknot-supported faults and normal faults. To achieve this goal, we statically disassemble and parse the kernel image and extract the address ranges of bowknot-supported functions and save them into a header file. When any exception occurs, we use this header file to execute our modified exception handler for bowknot-supported faults and execute the unmodified exception handler otherwise. In our modified exception handler, after setting `bowknot_flag`, before returning to the buggy function, we advance the Program Counter (PC) register to skip the crashing instruction. In ARM architecture, all instructions have the same length, and we simply advanced the PC register by four. However, x86 instructions have variable lengths. As a result, we need to decode the current instruction's length to advance the PC to the next instruction. We use Zydis for this purpose, which is a lightweight open-source disassembler library for x86 and x86-64 instructions implemented in C [15]. Since Zydis is implemented with no third-party dependency (not even libC), we can build Zydis as a part of the Linux kernel. To minimize code added to the kernel, we only port parts of the Zydis necessary to decode the instructions' length.

For ARM, we add 72 lines of C code and 42 lines of assembly code to the kernel exception handler. For x86, we add 136 lines of C code to the kernel exception handler and port 4677 lines of C code from the Zydis library.

3.6 Evaluation

3.6.1 Effectiveness

Effectiveness in Bug Mitigation

Methodology. To test the effectiveness of Hecaton and bowknots, we test our bug workaround against 113 bugs in Android and x86 Linux kernel consisting of real CVEs, unpatched real bugs, and injected bugs. Using a combination of real and synthesized bugs to evaluate the effectiveness of fault-tolerant systems is a common practice[144][151]. However, previous similar work, Talos[144], only used 11 real-world vulnerabilities and FGFT[151] tested no real-world bugs. In contrast, we use 39 real-world bugs. Similar to Talos and FGFT, to evaluate the effectiveness of bowknots, we measure two factors for each bug. First, whether the bug is successfully mitigated, and second, whether the system including the buggy module remains functional after the undo.

In our experiments, we use PoCs to trigger the bugs. In a successful mitigation, we make sure that the PoC still triggers the bug after bowknots insertion but that the execution of bowknots neutralizes the syscall that triggers the bug in a way that prevents the system from crashing, freezing, or generating further warnings by kernel sanitizers.

In addition, we test the functionality of the buggy module after the execution of bowknots as a result of triggering each bug. For our functionality test, we use standard benchmarking and self-test programs when they are available for a kernel module (e.g., GPU benchmarking application or Linux self-tests for a file system). Self-tests are small test programs that kernel developers have designed to exercise individual code paths in the kernel and report whether or not they achieve the expected outcomes. If no standard benchmark or self-test is available for a module, we manually test the underlying device of the buggy device driver in different

configurations (e.g., taking pictures and videos in different settings to make sure the camera is functional.)

For comparison, we also test and report mitigation and functionality preserving for each bug using Talos [144], which uses code disabling (§3.1.3). Since Talos disables parts of the code, it might seem unnecessary to test Talos workarounds for functionality. However, in some cases, the disabled function does not play a crucial role in the functionality of the device, for example, when the bug is located in a function that logs the device driver's events. In these cases, code disabling (Talos) might preserve the functionality of the device.

As we discuss in §3.7, bowknots cannot be used for the bugs located in the kernel's clean-up paths. Hence, we only measure and report (in §3.7) how common this limitation is, and we do not consider them in our effectiveness evaluations.

We also evaluate the effectiveness of Hecaton in generating complete bowknots. First, we report whether the bowknots get executed automatically or if we manually encode the condition for its execution. Second, we report whether the automatically generated bowknots are complete or if we manually add statements to complete them. For each bug, we limit the amount of manual effort to complete its bowknots to 2 hours. If we could not fix a bowknot manually in 2 hours, we record it as unsuccessful.

CVEs and Real Bugs in Android To evaluate the effectiveness of bowknots and Hecaton in mitigating real bugs and vulnerabilities of Android devices, we use 9 real bugs and reported CVEs in four kernel components of the Pixel3 smartphone: binder IPC, camera driver, GPU driver, and the TCP layer in the network stack (used with WiFi).

Table 3.1 shows the result. It shows that bowknots are effective in mitigating the bugs and vulnerabilities in 100% of cases and maintain the system functionality in 100% of these cases. 88.9 % of bowknots use automatic triggers and only one case uses manual triggers. Moreover, Hecaton is capable of generating complete bowknots in 55.6% of cases. In contrast, Talos

Kernel Modules	Bug/ Vulnerability	Talos Mitigate?	Talos Preserve Function?	Bowknot Mitigate?	Bowknot Preserve Function?	Bowknot Trigger Mode	Hecaton's Generated Bowknots
Binder IPC	CVE-2019-2215	✓	✗	✓	✓	Manual	Not-Complete
	CVE-2019-1999	✓	✗*	✓	✓	Automatic	Complete
	CVE-2019-2000	✗	✗	✓	✓	Automatic	Complete
Camera Driver	CVE-2019-2284	✗	✗	✓	✓	Automatic	Not-Complete
	Bug1**	✗	✗	✓	✓	Automatic	Not-Complete
	CVE-2019-2293	✓	✗	✓	✓	Automatic	Not-Complete
GPU Driver	CVE-2019-10529	✓	✗*	✓	✓	Automatic	Complete
	CVE-2018-5831	✓	✓	✓	✓	Automatic	Complete
Network (TCP)	CVE-2019-18805	✓	✓	✓	✓	Automatic	Complete

Table 3.1: *CVEs and real kernel bugs tested with bowknots. (* In these cases, the system was functional right after mitigation by Talos, but it stopped working after a while due to a memory leak resulting from code disabling, **Bug1: bug in msm_camera_power_down)*

Total # of tested Bugs	# mitigated by Talos	# function preserved by Talos	# mitigated by bowknots	# function preserved by bowknots	# automatic bowknot trigger	# complete bowknots by Hecaton	# added statements*
30	20	8	27	27	30	18	2

Table 3.2: *Unpatched bugs experiments (x86 Linux kernel bugs reported by Syzbot). (*Average # of added undo statements for incomplete bowknots by Hecaton)*

can only mitigate the bugs in 66.7% of cases and preserve the functionality in 22.2% of these cases. We discuss five of these vulnerabilities in Appendix.

Unpatched Real Bugs in x86 Linux kernel To further evaluate the applicability of bowknots and Hecaton to different targets and unpatched bugs, we use 30 real bugs in x86 Linux kernels reported by Syzbot [40]. We choose the 30 latest unpatched bugs (as of July 2020), which have reproducer PoC programs. The 30 bugs we test are located in various parts of the Linux kernel such as network stack, file system, memory management, HCI Bluetooth driver, and TTY driver.

Table 3.2 shows the results. It shows that bowknots are effective in mitigating the bugs and vulnerabilities in 90% of cases and maintain the system functionality in 90% of these cases. Moreover, Hecaton is capable of generating complete bowknots in 60% of cases. In contrast, Talos can only mitigate the bugs in 66.7% of cases and preserve the functionality in 26.7% of these cases.

Kernel Modules	# Injected Bugs	# mitigated by Talos	# function preserved by Talos	# mitigated by bowknots	# function preserved by bowknots	# automatic bowknot trigger	# complete bowknots by Hecaton	# added statements*
Camera	41	34	5	40	33	33	26	2
Binder	33	14	12	30	30	26	24	4

Table 3.3: *Bug injection experiments (camera device driver and Binder IPC).(*Average # of added undo statements for incomplete bowknots by Hecaton)*

device	driver	version	bugs	U. reboots	U. fuzz time	B. reboots	B. fuzz time
Pixel3	Camera	2018-08-22	3	1035 ± 60	$12h18m \pm 9m$	98.3 ± 114	$22h49m \pm 1h5m$
Nexus 5X	Camera	2016-10-13	6	622.3 ± 48	$12h10m \pm 19m$	12.0 ± 0.0	$23h19m \pm 1m$

Table 3.4: *Effective fuzzing time. U. and B. refer to using unmodified kernel vs. a kernel updated with bowknots. The number of reboots are per hour. Up time which is the overall time during which the fuzzer is running including wasted reboot time is 24h for all experiments. Fuzz time (i.e., effective fuzz time) is the time during which the fuzzer is actually fuzzing the kernel of the device.*

Injected Bugs in Android To further test the ability of bowknots in maintaining the system functionality, and test the robustness of Hecaton against the location of the bugs in the kernel functions, we use bug injection. More specifically, we inject 41 bugs in the camera driver of Pixel3 and 33 bugs in its binder IPC subsystem. To avoid any bias in favor of or against Hecaton, we randomly choose the bug injection location. To do so, first, we fuzz each module using Syzkaller to identify all lines of code reachable through the syscall interface. Next, after excluding the locations in the kernel's clean-up paths (see §3.7), we randomly choose one of the reachable lines and insert an explicit `BUG()` function there. Since the inserted `BUG()`'s location is random, an arbitrary number of state-mutating statements might get executed prior to the bug, which needs to be undone by a bowknot. As a result, this evaluates the ability of Hecaton in generating effective bowknots in various cases. We then generate bowknots using Hecaton and apply them for each bug. Table 3.3 shows the results. It shows that bowknots are effective in mitigating the bugs in 94.6% of cases and maintain the system functionality in 85.1% of these cases. Moreover, Hecaton is capable of generating complete bowknots in 70.4% of cases. In contrast, Talos can only mitigate the bugs in 64.9% of cases and preserve the functionality in 23.9% of these cases.

For all bugs for which Hecaton's bowknots were incomplete (injected bugs as well as real bugs and vulnerabilities), we needed to add on average 3 statements.

Effectiveness of Syscall Undo

We perform a detailed case study to evaluate bowknots' syscall undo capability. We perform a manual line-by-line investigation on the execution path of 10 real bugs (5 Android kernel and 5 x86 Linux bugs randomly chosen from the bugs discussed in §3.6.1). In this investigation, we search for any statement that changes the global state of the system but is not undone by bowknots. The result of this analysis shows that, to the best of our knowledge, for 6 cases the undo was complete and there were no changes to the system global state that did not get undone by the bowknots. Additionally, in 3 of the 4 failed cases, we could manually add the undo statements for the missed state-mutating statements and complete the bowknot in less than 2 hours. In the remaining one case, the state gets corrupted in a way that we even could not generate a complete bowknot manually. We discuss this case-by-case analysis in detail in the Appendix.

Effectiveness of Confidence Score

To evaluate Hecaton's confidence score, we use our corpus of 30 unpatched real bugs in x86 Linux kernel, which we discussed in §3.6.1. As mentioned in §3.4.3, Hecaton generates a confidence score for each bowknot instrumented function. Even if only one bowknot fails to undo the side effects of a partially executed function, the system state might remain inconsistent. As a result, to evaluate each bug, we consider the minimum confidence score for the bowknot instrumented functions in its call stack. We divide these 30 bugs into two sets of 20 and 10 bugs for respectively tuning and testing our confidence score. We tune the six coefficients of the confidence score (§3.4.3) in a way that it best separates the tuning set of

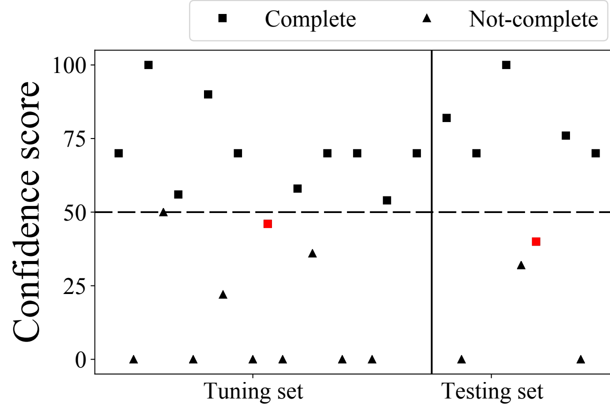


Figure 3.3: *Hecaton Confidence score prediction for Tuning and Testing sets*

bugs into two groups, one with complete bowknots and one that needs manual effort. Then we measure how well the tuned confidence score can predict the completeness of the bowknots Hecaton generates for 10 bugs in the testing set. Note that a false negative prediction is more acceptable than a false positive because in the case of a false negative the confidence score predicts an incomplete bowknot, which ends up being complete. Figure 3.3 shows that the confidence score works for 95% of the cases in the tuning set, and it predicts the completeness of generated bowknot with 90% accuracy in the testing set. Please note that there is no false positive in the results. In other words, whenever the minimum confidence score is greater than 50, the bowknots are complete.

3.6.2 Manual Effort for the Pair Database

We measure how much manual effort is needed to keep Hecaton's function-pair knowledge database updated with the ongoing updates in the kernel. For this purpose, we use Hecaton to generate the databases for 9 consecutive versions of x86 upstream Linux kernel, i.e., v5.0 to v5.8. As we discuss in §3.4.1, this database needs to be manually inspected and verified. Our measurements show that when we move from one kernel version to the next, on average 115 ± 18 additional function pairs need to be verified, which in our experience takes between 2 to 3 hours.

3.6.3 Performance Overhead

We measure the overhead of bowknots on the normal performance of the system. To do so, we measure how the performance overhead increases as the number of executed functions with bowknot instrumentation increases. To test the performance overhead of bowknots in our ARM implementation, we use two benchmark applications, "GPU Mark benchmark" that measures the output frame-rate of GPU renderings, and "Tamosoft Throughput Test" that measures the downlink TCP throughput. To test the performance overhead of bowknots in our x86 implementation, we use iPerf tool[8] in Linux kernel to measure the downlink TCP throughput.

Each benchmark results in the execution of many functions in their corresponding kernel components. First, we detect all these triggered functions (410 functions in the Pixel3 GPU driver, 390 functions in the Pixel3 networking stack, and 370 functions in x86 Linux networking stack). We then randomly choose a number of these functions and instrument them with bowknots. For all modules, we either instrument 100, 200, or all available functions in them. We run the benchmarks 10 times and show the average±stdev throughput in Figure 3.4.

The results show that there are no statistically noticeable performance drops even if all executed functions are instrumented with bowknots.

3.6.4 Use-Case Evaluation

As discussed in §3.1.2, by neutralizing bug-triggering syscalls, bowknots can help reduce the number of repetitive reboots during a fuzzing session. We evaluate the benefits of bowknots for fuzzing in this section. We fuzzed 13 device drivers and kernel components (camera driver, GPU driver, audio driver, WiFi driver, ION, Binder, and Ashmem) in three smartphones (Pixel3, Nexus 5X, and Samsung S7). Out of these, 5 of them showed repetitive reboots due

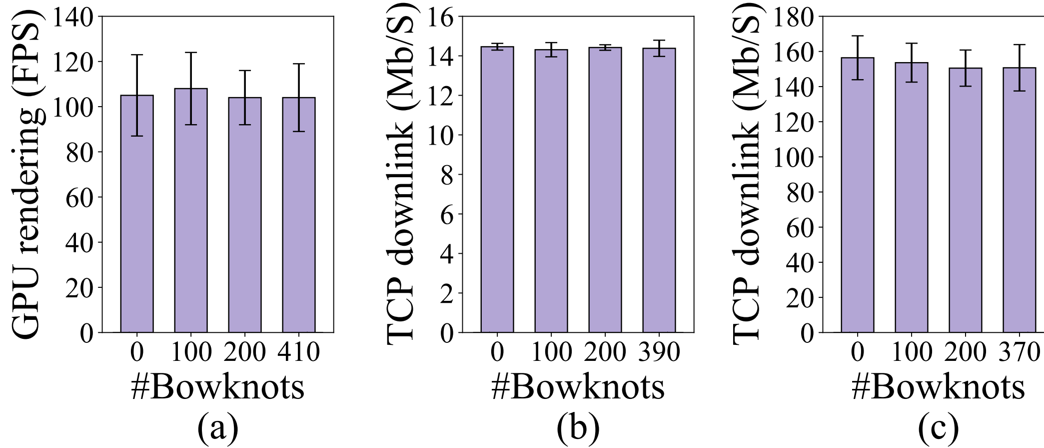


Figure 3.4: *GPU and TCP performance as the number of executed bowknots increase. (a) Pixel3 GPU , (b) Pixel3 TCP, (c) x86 upstream Linux (running in QEMU) TCP.*

to easily-triggered bugs. Out of these 5 drivers, 2 of them had easily-triggered bugs that bowknots could effectively mitigate. We show the results for these two drivers: the camera device driver of Pixel3 and the camera device driver of Nexus 5X. We note that bowknots cannot provide any benefits for the other three drivers.

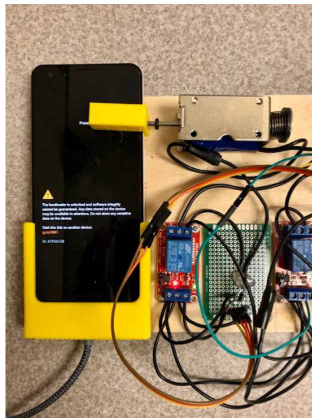


Figure 3.5: *The setup used in our fuzzing experiments.*

We use the following experimental methodology. We run each fuzzing experiment for 24 hours as suggested by Klees et al. [157]. Moreover, we repeat each experiment 3 times and report averages and standard deviations. To implement this methodology, we faced and solved a practical challenge. More specifically, running 24-hour kernel fuzzing experiments on smartphones proved to be challenging due to unreliability of the Android Debug Bridge (ADB).

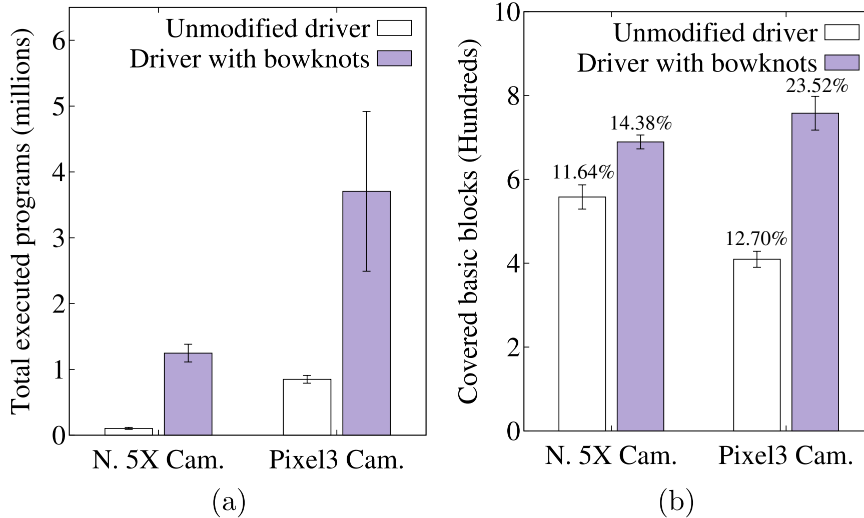


Figure 3.6: *(a) Total executed fuzzing programs. (b) Covered basic blocks (code coverage percentage is also reported on top of each bar).*

Occasionally, ADB would malfunction and the desktop machine running the fuzzer would lose its connection to the device, disrupting the experiment. This phenomenon happened more frequently when the device was rebooted more often. Our first attempt to address this problem was to restart the experiment from scratch when this issue happened. Given that experiments are 24 hours long, this proved to be a very lengthy process. Therefore, we built a custom hardware-software framework to programmatically and forcefully reboot the device using its power button when the connection to the device was lost. Figure 3.5 shows this setup. We 3D printed the cover to hold the smartphone in place, used a 45 Newton linear solenoid to press and hold the power button, and used an Arduino Uno board to control the solenoid from the fuzzer.

Increased fuzzing time. Table 3.4 shows the effective fuzzing time achieved when fuzzing the unmodified driver and the driver with bowknots. As the table shows, bowknots increase the effective fuzzing time by $88.6\% \pm 4.6\%$.

Executed programs. Figure 3.6a shows the total number of executed fuzzing programs. Bowknots eliminate wasted fuzz time and hence the fuzzer executes more programs. Our

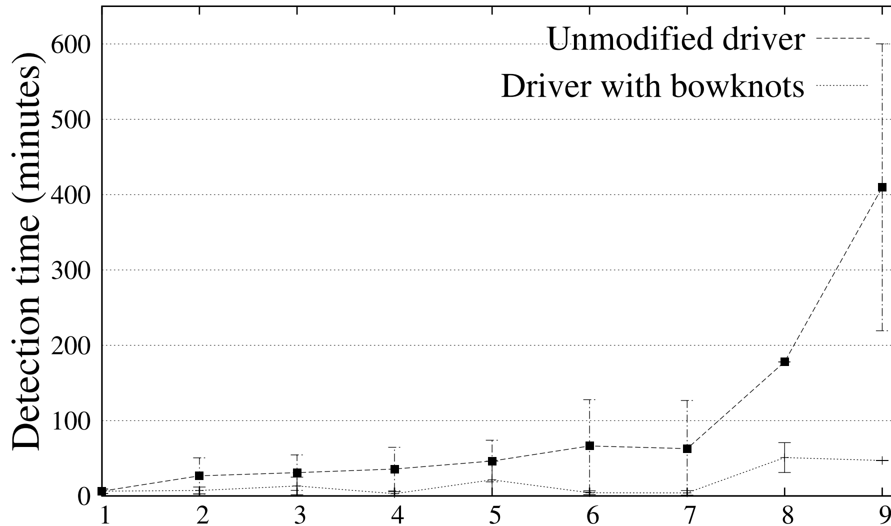


Figure 3.7: *Time taken for the fuzzer to discover a bug (i.e., trigger a bug for the first time). Each x-axis tick represents a unique bug.The points with no error bars represent bugs only found once during experiments*

results show that we manage to execute $723.5\% \pm 124\%$ more fuzzing programs on average with bowknots.

Code coverage. Figure 3.6b shows the code coverage in the driver under test. As can be seen, the higher number of executed programs and fewer reboots result in $54.3\% \pm 6.1\%$ higher code coverage.

Comparison with Talos. We compare the effectiveness of our approach in improving the fuzzing efficiency with Talos. To do this, we apply Talos to buggy functions in our fuzzing experiments. Our analysis shows that Talos, as a result, disables a large number of basic blocks, effectively lowering the code coverage. Moreover, our analysis shows that bowknots, when applied to the kernel, allow the fuzzer to cover a large part of the basic blocks that Talos disables. Table 3.5 shows the results. The results are insightful. Talos' approach disables the code unconditionally resulting in disabling 1290 basic blocks overall. However, bowknots only undo the syscall when they are triggered. Therefore, they allow the code

Bug triggered by fuzzer	Basic blocks disabled by Talos	Basic blocks disabled by Talos & covered by bowknots
msm_actuator_subdev_ioctl	141	129
msm_camera_io_w_mb	2	2
msm_camera_io_r	2	2
msm_flash_config	91	82
msm_csid_config	37	35
msm_cpp_subdev_ioctl	785	459
cam_ife_mgr_acuire_hw	71	45
cam_sensor_core_power_up	109	67
msm_camera_power_down	52	32

Table 3.5: *Bowknots vs. code disabling (Talos) for fuzzing.*

to be executed with good inputs, i.e., those that do not result in triggering the bug. This proves to be critical for achieving good code coverage when fuzzing. As a result, bowknots help cover 66% of the basic blocks disabled by Talos.

Faster and more effective bug detection. By eliminating reboots with bowknots, we manage to find bugs faster. Figure 3.7 shows the list of all the bugs found in the two drivers. It shows on average the time it takes to find the bug in drivers with and without bowknots. Bowknots help us find all these bugs faster. On average, we find the same bugs faster by 42.6 minutes. This speed-up varies between 6 minutes to 162 minutes for different bugs.

3.7 Other Limitations

Undetected corruptions. Bowknots' effectiveness depends on catching the errors before they corrupt the system and undo the effect of the system call that causes the error. In some cases, a crash as a result of a bug (e.g., out of bound write/read to/from a non-allocated address) triggers the execution of bowknots. However, in cases that the same bug does not result in a crash, bowknots rely on kernel sanitizers (e.g., KASAN) to catch the error before it corrupts the kernel. In cases where there is no crash, kernel sanitizers do not catch the error, or they are not enabled in the kernel for performance reasons, the analyst needs to

provide the check for triggering the bowknot, otherwise the bowknots might not be secure and effective.

Bugs in clean-up paths. Bowknots are designed based on the idea of undoing the effect of partially executed syscalls. However, undoing the effect of syscalls that are themselves designed for clean-up is not possible. Consider a syscall designed to destroy a few kernel objects and free all the allocated memories. If a crash happens in the middle of this syscall, where half of the kernel objects are destroyed, no bowknot could re-create the exact objects and undo the effect of this partially executed syscall. We studied the latest 100 bugs of Linux upstream kernel reported by Syzbot (as of October 2020). Our study showed that 28% of the bugs are located in clean-up paths and hence were not amenable to bowknots.

Chapter 4

MegaMind: A Platform for Security & Privacy Extensions for Voice Assistants

Voice assistants, such as Amazon Alexa [45], Google Assistant [57], Apple Siri [67], and Microsoft Cortana [65], are becoming ingrained in our personal lives. Beyond their prevalent integration into smartphones and tablets, they are now increasingly found in home speakers [46, 59, 62, 60], cars [24, 36, 35], children's toys [33], light bulbs [30], TV sets [34, 25], and other appliances. Several companies have even released device SDKs to simplify adding voice assistance to any hardware device [49, 58, 50].

Voice assistants provide a convenient user interface: natural language. However, this convenience comes with serious security and privacy risks. A voice assistant uses an always-on microphone and operates by capturing audio and sending it to the manufacturer's cloud service for processing. The cloud service transcribes the audio and interprets it as user requests. Audio recordings can have private and sensitive content, such as medical or sexual

information [140]. Moreover, interpreted requests may result in unintended or unapproved actions, such as a purchase or a phone call. These unintended actions can be either due to "mistakes" by the assistant, or attacks [32, 31]. Moreover, the assistants' responses might contain inappropriate content, such as content not suitable for children [21].

To make matters worse, voice assistants incorporate many third-party applications, i.e., *skills*[1], which enhance the assistant functionality [69]. Unlike mobile apps, skills do not run on the voice assistant hardware. Instead, they are cloud services invoked by the manufacturer's cloud service. Researchers have shown a plethora of additional security and privacy concerns surrounding third-party skills [118, 185, 234, 94], including malicious skills [161] and unintended voice data leaks [28, 119].

This chapter presents *MegaMind*, a security and privacy extensibility platform for voice assistants. MegaMind extensions execute locally on the assistant itself. They intercept the recorded audio before sending it to the manufacturer's cloud service, and the response audio before delivering it to the user. Extensions can thus inspect, modify, or discard unwanted content to meet a user's security and privacy goals. For example, a redaction extension removes any mentions of a user's personal information from the recorded audio.

MegaMind's design enables novel extensions that bring a level of unprecedented security to users. For example, we implement *secure conversation*, an extension that provides end-to-end encryption, integrity, and rollback protection to let a user conduct a secure conversation with a trusted skill such as a bank, without the voice assistant manufacturer having unrestricted access to the conversation. As another example, we implement *anonymous query*, an extension that employs a mixer cloud service to enable a user to remain anonymous (to the voice assistant manufacturer and to third-party skills) when issuing sensitive queries such as medical queries.

[1] *Skill* is Amazon's Alexa service terminology for voice assistant apps, but we use it broadly for all assistants.

MegaMind does not blindly trust extensions and assumes they can be malicious. To protect users from such extensions, MegaMind provides two security guarantees, *permission enforcement*, and *non-interference*. Permission enforcement limits the conversations each extension can access (i.e., *access permissions*) and modifications it can perform on them (i.e., *modification permissions*). Moreover, *non-interference* guarantees that a malicious extension cannot modify the conversation in a way that disrupts the execution of other extensions.

Given the richness of natural language, providing such guarantees is challenging and requires a careful design and a comprehensive security analysis. To do so, MegaMind provides a novel programming model for extensions. Each extension consists of a *manifest* and an *action function*. In the *manifest*, an extension declares its needed permissions using MegaMind's easy-to-review and straightforward permission description language. *Action functions* are generic Python scripts that process the phrases. We enforce several limitations on what an extension can do, and we demonstrate, with a careful analysis, that our design provides the aforementioned security guarantees.

Any third-party can develop MegaMind extensions. In addition, MegaMind can provide an extension market (similar to the application market for smartphones) in which developers publish their extensions. Similar to application markets, a MegaMind extension market can audit all extensions prior to publication, and reject those with malicious manifests. In addition, the extension market can authenticate the extension developers. For example, if a companion extension for a third-party skill implements secure conversation, then MegaMind can easily check that this skill and its companion extension are published by the same entity.

We build MegaMind and integrate it with the Amazon Alexa Service SDK. Thus, it is potentially deployable on many commercial devices that use the Alexa SDK, such as the Acer Spin 5 Convertible Notebook [43] and the Fitbit Versa 2 smart-watch [71]. MegaMind is also compatible with all Alexa skills. To minimize conversation latency, we optimize MegaMind's implementation in several ways, such as using a sandbox pool to reduce startup

latency. Our prototype is mature, allowing users to have multi-turn conversations with Alexa Voice Service (AVS) or third-party skills. We open source the prototype for the benefit of users and researchers and provide a video demo showing MegaMind's performance and novel extensions.[2]

We evaluate MegaMind's implementation on three hardware platforms: two ARM SoC platforms, Raspberry Pi 4 (RPi 4) and Raspberry Pi 3 (RPi 3), and an x86-based laptop. Our ARM prototypes represent lower-end mobile devices such as smartphones, modestly-powered standalone assistants, and embedded ones. Our x86 prototype, on the other hand, represents higher-end and more powerful assistants. MegaMind achieves good performance on the RPi 4 and on the laptop, but suffers from high performance overhead on the weaker RPi 3. This performance discrepancy is expected; MegaMind has moderate local processing needs, including speech-to-text conversion and NLP processing, and the RPi 3 processor is not powerful enough to meet these needs [41]. Nevertheless, our evaluation shows that MegaMind's processing requirements can still be met by an inexpensive platform such as the RPi 4. We also perform extensive testing to evaluate MegaMind's ability to deliver on its security and privacy goals using a large corpus of sample conversations. Our results show that MegaMind achieves high accuracy (less than 10% false positive and false negative rates) in many cases, although it can sometimes experience lower accuracy. Our investigation shows that local speech-to-text conversion is an important contributor to MegaMind's inaccuracy. We expect that future conversion engines will further improve MegaMind's effectiveness.

We make the following contributions.

- We present the first extensible platform for enhancing the security and privacy of voice assistants.
- We design a programming model for extensions that enables ease of development and

[2] `https://trusslab.github.io/megamind/`

high expressibility.

- We design an extension execution framework that provides permission enforcement and non-interference guarantees for the extensions.
- We demonstrate novel extensions for voice assistants, including extensions for secure conversation and anonymous query. These extensions provide security guarantees not possible in today's voice assistants.
- We perform several optimizations in our prototype to achieve a low conversation latency overhead, critical for the adoption of MegaMind in practice.
- We perform an extensive evaluation of MegaMind and show that it incurs a small conversation latency overhead, has modest CPU utilization, and is effective in achieving its security and privacy goals.

4.1 Motivating Extensions

MegaMind enables many security and privacy extensions. In this section, we describe extensions we have developed and tested.

Secure conversation. A user may want to converse in a secure manner with a trusted third-party skill, such as a bank or a health provider skill. The user may want to protect the conversation detail both from other third-party skills and from the voice assistant cloud service, e.g., AVS. In §4.6.1, we discuss how an extension can provide end-to-end encryption, integrity, and rollback protection for such conversations. MegaMind lets a user send and receive ciphertext over AVS, a novel functionality not demonstrated before.

Anonymous queries. A user may want to issue a sensitive query without revealing their identity. The user does not want the assistant cloud service or any third-party skills to

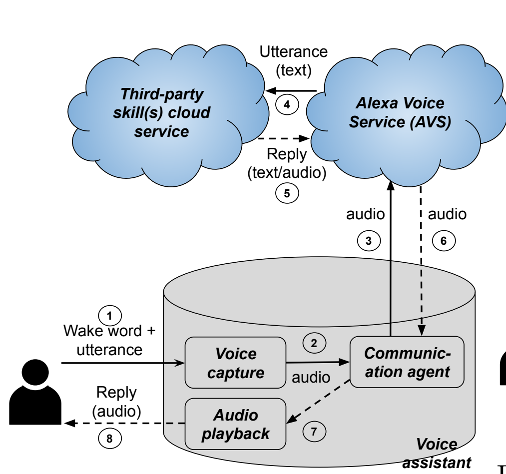


Figure 4.1: *Amazon Alexa voice assistant architecture.*

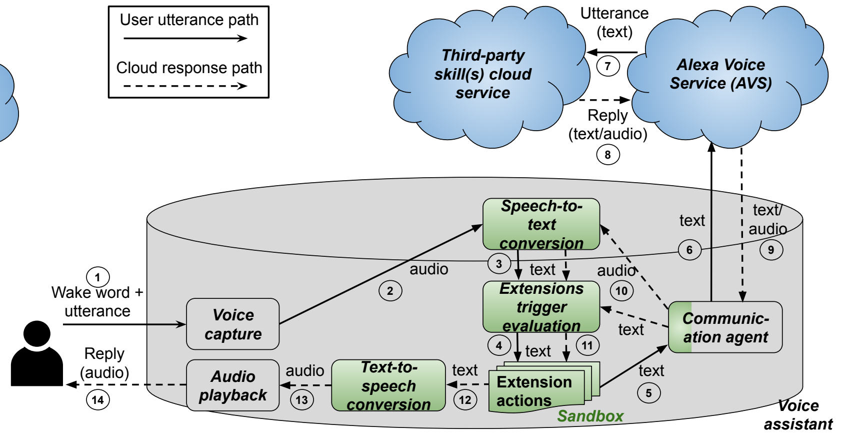


Figure 4.2: *Adding the MegaMind extensibility platform to Amazon Alexa. MegaMind's functionality is shown in green.*

associate the query with them. For example, a user may want to issue a medical query anonymously to protect the user's underlying medical conditions. In §4.6.2, we show how a MegaMind extension along with a mixer skill can realize this novel security feature.

Redaction. This extension's goal is to protect sensitive user information, such as family members' names, phone numbers, or credit card numbers, from being revealed to AVS or third-party skills.

Night mode. A user may want to disable an assistant placed in a certain location (e.g., bedroom) during a certain time period (e.g., 10 PM to 7 AM).

Parental control. A user may want to enforce access control policies when a voice assistant is used by their children. For example, they might want to limit access to certain skills or limit usage time periods. Moreover, they may want to block any form of purchases and prevent assistant's responses from including adult content, violent content, or profanity.

Phone call control. A user might want to block the assistant from making calls to phone numbers outside a contact list. This can prevent unintended and malicious calls triggered by the voice assistant [32, 38, 37].

Third-party skill limiter. A user might want to limit the set of third-party skills their assistant can communicate with. Such an extension would help mitigate voice squatting attacks [161].

Please note that these are only a few examples of extensions that MegaMind enables. MegaMind's programming model and permission system allow the development of various extensions capable of performing edge computing in a controlled and secure manner.

4.2 Architectural Overview

This section presents an overview of MegaMind's architecture. For simplicity and without loss of generality, our description of MegaMind is based on Amazon Alexa voice assistants.

Figure 4.1 illustrates the interactions among users, their voice assistant, AVS, and third-party skills in existing commodity voice assistants. A wake word, such as *"Alexa"*, invokes the assistant to start recording a user's *utterances* and send them to AVS. AVS parses the audio and interprets it as a user request. Note that some captured utterances could be the result of accidental [32, 31, 38] or malicious [94, 262, 208] wakings of the assistant rather than intended user requests. AVS handles the request internally (i.e., using *built-in skills*) or sends it to a third-party skill for processing.

In addition to one-shot queries (i.e., a question and an answer), the voice assistant may enter a *dialog mode* that consists of multiple questions and answers, forming a *multi-turn conversation* [77]. Dialog mode's goal is to gather and confirm all the information needed for servicing a user request. For example, when ordering an Uber, AVS may ask about the type of the ride, the number of passengers, and the departure time. Hereafter, we refer to all requests and responses in a conversation as one *session.* In each session, the user interacts with either a built-in AVS skill or with a third-party one.

Figure 4.2 shows the voice assistant architecture when incorporating MegaMind. MegaMind interposes on communications from the voice assistant device to the voice assistant cloud service. It converts the user's utterance to text, evaluates it against the *trigger rules* of deployed extensions, and invokes the extensions' *action functions* on a trigger rule match. To protect against malicious extensions, MegaMind provides permission enforcement (§4.4) and non-interference (§4.5) to limit what an extension can do. Once processed, the text-based utterance is sent to AVS.

On receiving a response, MegaMind's deployed extensions process it before sending the possibly modified response for audio playback. The response from AVS can be in audio or text formats. If in text format, the communication agent directly sends it to be evaluated by the extensions. If not, the audio is first converted to text. Finally, the response needs to be converted back to audio for playback for the user. MegaMind achieves this using a text-to-speech converter.

The figure also shows that MegaMind executes the action functions of extensions in sandboxes (§4.7.1).

4.3 Trust & Threat Model

There are seven participants in MegaMind's ecosystem: 1) the owner of the device, 2) the user of the voice assistant (which might be the owner or someone else in owner's household such as their children) 3) voice assistant cloud service provider and its vendor (AVS/Amazon in case of Alexa), 4) the vendor of the voice assistant hardware, 5) third-party skills, 6) MegaMind, including all its software components, and 7) extensions, including (7A) extensions' manifest and (7B) extensions' action function. We note that (4) can be different from (3). For example, Amazon allows third-party hardware developers such as Sonos to

build voice assistant devices that use its voice assistant cloud service. Even users can deploy Amazon's open-source Alexa SDK on their personal computers. Therefore, (4) provides the voice assistance hardware and the system software (OS and firmware) running on it.

We develop MegaMind's threat model from the perspective of the device owner (1). We assume that the owner always trusts the voice assistant hardware vendor (4) and MegaMind (6). The owner does not trust the MegaMind third-party extensions' action functions (7B). Moreover, we assume the owner reviews the installed extensions' manifest (7A) and ensures it does not ask for malicious permissions. Hence, we suggest that extension developers make the manifests publicly auditable. We also suggest that extensions developers sign extensions' manifest and action functions. This signature can be checked before installing the extension to verify its authenticity. Unless otherwise specified, the owner does not trust users (2). Users may accidentally share their private information or intentionally perform actions that the owner does not authorize.

In addition to the mentioned trust model, which applies to all the extensions, we discuss the trust model specific to different goals each extension tries to achieve.

1. Privacy protection. These extensions prevent users from accidentally sharing their private information with AVS and/or skills. For these extensions, the owner does not trust AVS (3) and third-party skills (5).

2. Content control. These extensions prevent AVS and skills to send sensitive responses to users. For these extensions, the owner does not trust AVS (3) and third-party skills (5).

3. Action limiting. These extensions prevent users from conducting some actions. For example, the owner might deploy one extension to block all purchases for the device installed in their child's room. For these extensions, the owner needs to trust AVS (3) or third-party skills (5) as they can conduct such action without the device sending them the request anyway.

4. Skill allow-listing and deny-listing. The owner uses these extensions to selectively trust a subset of skills. Using these extensions, the owner can either block the usage of a subset of skills or selectively apply privacy preserving or content control on them. For these extensions, the owner trusts AVS (3), and trusts/distrusts a subset of third party skills (5). Please note the owner needs to trust AVS since it is in charge of routing the commands to the third-party skills.

5. Skill security enhancement. The owner uses these extensions to provide a security feature for one specific third party skill. These extensions use secure channels to protect the content of the conversation from AVS. Using this extension, the owner trusts only one third party skill and its companion extension on its assistant.

Adversarial model. We assume that the adversary has full control over untrusted participants other than the user. For example, when we assume that AVS is untrusted, we assume that it is controlled by the adversary. When the user is untrusted, we assume they can make mistakes, but they do not act maliciously. Side-channels attacks, physical attacks, and any other attacks that allows the adversary to compromise a trusted participant are out of this work's scope.

Defense against attacks. Several important attacks have been demonstrated on voice assistants. *Inaudible voice attacks (IVA)* and *concealed voice attacks (CVA)* stealthily deliver voice commands to a voice assistant without the user knowing. [207, 262, 208]. Similarly, research projects on concealed voice commands showed that devices continue to respond to wake words and utterances even when "mangled" to such a degree that they are unintelligible to users [234, 94]. Another important attack is the *voice squatting attack (VSA)*, where a malicious skill developer creates an invocation phrase similar to a legitimate skill, in the hope that sometimes the wrong skill may be invoked and data may leak [161, 264]. Another important attack is the *fake skill termination attack (FSTA)*, [264], where a malicious skill developer creates a long silent audio response in order to trick the user into thinking that

Attack category	Examples	How?
Phishing	VSA	Skill deny-listing
Eavsdropping	FSTA, CVA	Privacy protction
Unautorized cmd	IVA, CVA	Action limiting

Table 4.1: *MegaMind protection for attacks.*

no skill is running anymore, at which point the user may say something private. Table 4.1 shows how MegaMind extensions help in protecting against attacks.

4.4 Permission Enforcement

Since MegaMind extensions are developed by untrusted third parties, we need to enforce limitations on these extensions. MegaMind provides a permission system similar to the Android permission system for applications. The owner reviews permissions an extension asks for and installs it only if she approves the permissions.

Despite similarities, MegaMind's permission model is fundamentally different from Android's. Android's permissions help limit the application in accessing different systems resources such as I/O peripherals. In contrast, MegaMind's permission system divides the permissions of each extension in two categories, *access permissions* and *modification permissions*. Access permissions limit the conversations an extension can mediate, and modification permissions limit how they can modify them.

Limiting the extensions' permissions is not trivial and requires special care. Some extensions, such as security enhancement extensions require to arbitrary change the phrases. However, it is not secure to give all extensions this permission. Also, it is not secure to allow a security extension to access and modify all the communications. As a result, we devise a permission model that strikes a balance between access and modification permissions. In our permission model, extensions that have more access permissions have more restricted

modification permissions. Based on this model, we divide extensions in three different types: discarders, sanitizers, and companions. We define each of these types and their permission model in the §4.4.2.

In the rest of the section, we first describe how an extension expresses its access permissions through trigger rules. We then discuss how extension types bring a balance between access and modification permissions.

4.4.1 Access Permissions

Every extension declares its required permissions in its manifest, a JSON formatted file consisting of an extension type and a rule-set (i.e., an array of *trigger rules*) (Figure 4.3). Trigger rules indicate an extension's access permission by specifying the utterances or responses (i.e., jointly referred to phrases hereafter) the extension needs to process. MegaMind provides a description language that makes it easier to declare trigger rules in a generic and transparent fashion. It facilitates reviewing the manifest to find malicious permissions. Besides, in §4.4.2, we show how MegaMind helps in preventing malicious access permissions by limiting the trigger rules an extension can use based on the extension's type.

Trigger Rules Description Language Each extension expresses its access permissions in a rule-set. MegaMind evaluates the rules of the rule-set against each phrase and, *if any of the rules evaluates to true*, MegaMind executes the action function on all subsequent phrases of the current *session*. A rule itself is a set of *conditions* on *keywords*, *time*, or *third-party skill ID*. These conditions are grouped in two sets: (1) an inclusive disjunction (shown with the predicate *include_or* in Figure 4.3), and (2) an exclusive conjunction (shown with the predicate *exclude_and*). A rule evaluates to true *if all of its conditions are true.*

Our trigger rules description language is expressive because it allows arbitrary trigger logic

using its language constructs. Using this language, an extension can request access permissions to phrases containing or not containing certain words, occurring in certain time periods, and belonging to conversations with certain skills. A MegaMind rule-set is the sum of products (SoP) of the conditions on keywords, time and skill ID, and any arbitrary logic can be expressed as an SoP.

NLP Helper Functions Although MegaMind's trigger rule description language is expressive, when using a natural language, constructing a set of conditions using keywords alone is challenging. Consider an extension that blocks adult content from responses returned by third-party skills. It is difficult to construct a comprehensive corpus of adult content using keywords alone. To improve MegaMind's expressiveness and ease of use, we provide *Natural Language Processing Helper (NLP Helper) functions*. Examples of functions provided in our prototype look for synonyms, antonyms, first and last names, phone numbers, addresses, violent content, adult content, and profanity. Third party extensions can only use the NLP helper functions and cannot modify or train them. Thus, they cannot use them as an attack surface to increase their access permissions.

Secure Skill ID Detection Skill ID is a crucial factor in determining access permissions of an extension. Hence, MegaMind needs to associate each phrase to a target skill in a secure manner. MegaMind uses two methods for skill ID detection: an *AVS-dependent method* and a *local method*. We use the former when AVS is trusted and the latter when it is not (e.g., for extensions that provide skill security enhancement, i.e. companions; see §4.3).

For the AVS-dependent method, we leverage the fact that AVS tags each response with the ID of the skill that provided it. Therefore, we use this AVS's metadata to associate a skill ID to a phrase.

For the local method, we try to detect skill invocations locally by analyzing the user's utterances. AVS establishes a conversation session between a user and a third-party skill in

```
{
 "type": "Extension Type", // discarder, sanitizer, or companion
 "rule_set": [
   // rule1
   {
     "keywords": {
       "include_or": [
         "contain(word 1)",
         "adult_word()", ... // can add more included words
       ],
       "exclude_and": [
         "synonym(word 2)", ... // can add more excluded words
       ]
     },
     "time": {
       "include_or": [
         { // time range 1
           "start": "start time 1",
           "end": "end time 1"
         }, ... // can add more included time ranges
       ]
     },
     "Skill_ID": {
       "exclude_and": [
         "skill ID 1", ... // can add more excluded skill IDs
       ]
     },
   }, ... // can add more rules
  ]
}
```

Figure 4.3: *Trigger rules description language.*

two ways: explicit invocation and implicit invocation. An *explicit* invocation is when the user deliberately invokes a skill using a specific grammar, e.g., "Open Uber". This grammar is deterministic and known by users and AVS [70]. An *implicit* invocation occurs when AVS delegates handling a request to a skill without the user asking. The implicit invocation only occurs with skills that implement the *name-free skill invocation* feature [70]. Our local method can only detect explicit invocations. This is because the grammar of an explicit skill invocation is known but there is no specific grammar for implicit ones. As a result, third-party skills with companion extensions (that need to rely on MegaMind's local method for their security) cannot (and should not) implement the name-free invocation feature.

E2 \ E1	User's requests						Assistant's responses					
	Discarder		Sanitizer		Companion		Discarder		Sanitizer		Companion	
	UP	OP	UP	OP	UP	OP	UP	OP	UP	OP	UP	OP
Discarder	✓(E)	✓(E)	✓(E)	✓(E)	✓(E)	✓(E)	✓(E)	✓(E)	✓(E)	✓(E)	✓(E)	✓(E)
Sanitizer	✓(O)	✓(L)	✓(E)	✓(E)	✓(O)	✓(E)	✓(O)	✓(L)	✓(E)	✓(E)	✓(O)	✓(E)
Companion	✓(T)	✓(O)	✓(T)	✓(O)	✓(E)	✓(E)	✓(O)	✓(T)	✓(O)	✓(T)	✓(E)	✓(E)

Table 4.2: *This table summarizes all possible types of interference extension E1 can cause on extension E2's execution. "✓" means MegaMind can prevent interference. In each case, interference is avoided by: Extensions' definition (E), Order of execution (O), Limitations on extensions (L), and Trust model (T).*

4.4.2 Modification Permissions

When a phrase triggers a rule, MegaMind invokes the extension's action function to process the phrase. It is not secure to permit extensions to modify the phrases arbitrarily. The set of modifications MegaMind allows an extension to make depends on the extension type, as described next.

Discarder. A discarder's action function can drop a phrase from the conversation but cannot modify it. Whenever a discarder extension drops a phrase, MegaMind notifies the user by saying: "The last phrase is dropped by [discarder's name] extension." This notification prevents a malicious discarder extension from stealthily dropping a sensitive phrase.

Given the limited modification permissions the discarders have, we give them broad access permission by setting no limitation on their trigger rules. Discarders are useful for action limiting and skill deny-listing goals.

Sanitizer. A sanitizer's action function is allowed to modify the phrase, however, not arbitrarily. MegaMind enforces two constraints on sanitizers' modification permissions: (1) it can change only the words within a phrase that its triggered rule specifies under the *keywords* label (including those detected by NLP helpers), and (2) its replacement words cannot be arbitrary but must be drawn randomly by MegaMind from a specific category, such as a first or last name, a phone number, a city name, a country name, a random N-

digit number, or a bleep censor (i.e., the word *bleep*). In §4.5, we discuss how this second limitation is also critical to MegaMind's guarantee of non-interference.

As sanitizers have more modification permissions, we limit their access permissions. Sanitizers cannot use *exclude_and* in the keyword section of their rule-set. This prevents sanitizers from excluding a rare category of words and getting triggered on many generic phrases. Sanitizers are useful for privacy protection and content control goals.

Companion. A companion extension is paired with a specific third-party skill. Therefore, its rules must list only one third-party skill ID that the extension accompanies. A companion's action function is allowed to make arbitrary changes to phrases. A third-party skill can have a single companion extension only. Companion extensions have strong modification permission, but their access permission is heavily limited. They are suitable for implementing extensions for the skill security enhancement goal.

4.5 Non-interference

In some cases, the same phrase needs to be processed by more than one extension. For example, consider two extensions, one that redacts the names of people living in a household in all conversations, and one that implements secure conversation with a third-party skill. Both extensions require processing the conversations between the user and that third-party skill.

In MegaMind, the extensions operate on phrases sequentially. This is because parallel processing would undoubtedly result in conflicts in the output that cannot be trivially resolved. For example, merging the encrypted output of one extension with the redacted output of another is impossible.

Since the extensions process phrases one by one, their execution's order can affect the final output. We develop a specific ordering for extension execution, which prevents interference. Our proposed extension execution orderings (which we justify next) are:

User's requests: sanitizers execute first, followed by discarders, and then followed by companions.

Assistant responses: companions execute first, followed by sanitizers, and then followed by discarders.

4.5.1 Non-interference definition

We show that our proposed orderings guarantee non-interference defined by the following two criteria: 1) No *under-protection (UP)* and 2) No *over-protection (OP)*. Below, we define UP and OP based on the notion of *protection actions*. Protection actions are: discarding the whole phrase, sanitizing words in the phrase, or securing the phrase (which might involve any arbitrary changes by the companion.)

UP occurs when an extension: 1) undoes the effect of a previously executed protection action, or 2) prevents the later extensions from performing their protection actions. As an example for (1), consider two extensions, a companion and a sanitizer that redacts the adult content from the assistant responses. If the companion executes after the sanitizer, it can add the adult content back to the phrase. As an example of (2), consider a companion that encrypts the user requests. If the companion executes prior to a privacy protecting extension, which redacts the user's personal information, the encryption hides the private information and makes the privacy protecting extension useless.

No OP means if an extension performs any protection action on a phrase, it would have performed the same action to the original phrase. As an example of an OP, assume a scenario

that a sanitizer extension runs before a discarder extension. If the sanitizer modifies a word in a phrase to a word forbidden by the discarder, the discarder will block that phrase. However, the discarder would not have blocked the original phrase.

Please note that the term "non-interference" might have been used with other meanings in other contexts and research fields. Any reference to non-interference in this paper refers to the above definition.

4.5.2 Non-interference guarantee

We list all possible interference types between extensions in Table 4.2. This section discusses how MegaMind provides a non-interference guarantee by preventing all these types. MegaMind prevents interference in four different ways: (1) access/modification permission limitations each **E**xtension has by definition, (2) **O**rder of execution, (3) extra **L**imitations MegaMind enforces on extensions to guarantee non-interference, and (4) the MegaMind's **T**rust model.

Below we discuss how MegaMind successfully prevents interference for every entry in Table 4.2 using one of the four ways mentioned above. In following paragraphs, we use `E1-E2-{req,resp}-{UP,OP}` naming convention to point to each entry in Table 4.2. We use '`*`', and '`{}`' to point to more than one entry. For example, `{dis,san}-*-*-UP` means UP interference a discarder or a sanitizer can cause on all extensions while processing user's requests or assistant's responses.

Extension Definition (1) Discarder extensions do not modify the phrases. As a result, they cannot cause any interference to other extensions (i.e. no `dis-*-*-*`). Please note that MegaMind's guarantee of non-interference does not protect against denial-of-service. A malicious discarder can drop all the messages. However, MegaMind will notify the user every

time the discarder drops a message, and the user will uninstall the malicious discarder. (2) MegaMind only allows one companion extension per skill. As a result, the `comp-comp-*-*` interference never happens. (3) No other extension can modify the skill ID of a session. Hence, they cannot cause the companion to process a message with wrong skill ID (i.e., no `*-comp-*-OP`). (4) As discussed in 4.4.2, sanitizers cannot use the exclusion of keywords in their trigger rules. It means a specific word (or word category) should be present in the phrase to trigger a sanitizer. Also, they can only change those words to a random word drawn by MegaMind. Together, these two limitations prevent a sanitizer from re-changing a word previously changed by another sanitizer (i.e., no `san-san-*-UP`). The same limitations prevent a sanitizer from changing a word to cause triggering of another sanitizer (i.e., no `san-san-*-OP`).

Order of Execution (1) Sanitizers run before discarders; thus, they cannot undo the protections discarders provide and cause UP (i.e. no `san-dis-*-UP`). (2) Companions run immediately before sending users' requests to AVS and immediately after receiving AVS responses. As a result, sanitizers cannot compromise the integrity or confidentiality of phrases going to/from AVS. (i.e., `san-comp-*-UP`) (3) Companions run after discarders and sanitizers for user requests. Hence, there is no way for them to cause OP on discarders or sanitizers (i.e. no `comp-{san,dis}-req-OP`). (4) Companions run before discarders and sanitizers for assistant responses. Hence, they cannot undo the protection actions provided by discarders or sanitizers. (i.e. no `comp-{san,dis}-resp-UP`)

Extra Limitations (1) Without further considerations, a sanitizer can cause OP interference with a discarder (i.e. `san-dis-*-OP`). Consider a sanitizer that redacts personal phone numbers and a discarder that blocks all the calls to phone numbers outside of a contact list. In this case, when the user tries to call a number on his contact list, first the sanitizer changes it to calling a random number, then the discarder blocks the call. However, if we passed the original phrase to the same discarder, it would not block it. Hence, it is a case of

OP.

In the mentioned case, the discarder and the sanitizer have an inherent conflict, and no order can result in their interference-free execution. MegaMind resolves this issue by identifying and preventing these conflicts at the time of extension installation. The rule we enforce is that a sanitizer and a discarder can have overlapping inclusion and exclusion keyword lists only if they work on a non-overlapping set of skill IDs. In the above example, the conflict would be resolved if the sanitizer extension makes an **exception** for a phone call skill and the discarder blocks phone calls **only** for that skill.

Trust Model (1) Companion extensions run after discarders and sanitizers for user's request with unlimited modification permission. Therefore, they can potentially undo the protection actions of discarders and sanitizers and result in UP (i.e., `comp-{san,dis}-req-UP`). (2) Because companions run before discarders and sanitizers for assistant responses, they can add keywords to the phrase which provoke discarders and/or sanitizers and cause OP (i.e., `comp-{san,dis}-resp-OP`).

However, a companion extension only runs when the user is conversing with its accompanied skill, and based on our trust model; it is as trusted as the skill. Companion is the last extension that processes the user's request before sending them to the skill, and it is the first extension that processes the incoming responses. The companion executes in an isolated sandbox; the only data it can access is the phrases in the ongoing conversation. Hence, whatever action the companion extension does could have been done by the skill itself. Thus, we do not consider these actions as interference with other extensions.

4.6 Novel Security Features

MegaMind enables novel security features for voice assistants. In this section, we provide more details on two such features.

4.6.1 Secure Conversation

This extension lets a user conduct a secure conversation with a trusted third-party skill. The whole conversation is encrypted, integrity- and rollback-protected, ensuring no intermediary including AVS have access to the conversation's plaintext nor can they tamper with the conversation's contents.

Workflow. Assume a user intends to converse securely with a bank skill named *Great Bank*. The user invokes the skill by a phrase like "Alexa, Open Great Bank". At this point, the extension gets executed on the voice assistant. Its first task is to share a symmetric *session* key with the skill. To do so, it generates the key, encrypts it with the skill's public key and waits for the response from the skill. The skill responds to the "Open Great Bank" message with a welcome message and asks the user if they want to establish secure communication to this skill. If the user answers "yes", the extension replaces the user's answer with "key is [encrypted key]" and sends it to the skill. AVS delivers the encrypted key to the companion skill. Since the key is encrypted with the public key of the skill, skill decrypts it with its private key. Now both sides have the same symmetric key. The session key is then used for encryption and integrity protection (HMAC). The skill sends all the messages back to the user encrypted using this symmetric key. The extension also, encrypts all the user's utterances with the session key and sends a message as follows for every utterance: "search for [ciphertext]", where "search for" is a carrier phrase, described later in the skill support subsection. In addition, messages include a monotonic counter value to prevent rollback.

The endpoints also check that the counter value is incremented with no gaps to ensure no messages are dropped. Finally, session tear-down is done explicitly using an *end-of-session* exchange to ensure the endpoints received all messages. If any of these checks fail, the session terminates with an error and MegaMind instructs the user to try again.

Encoding. Voice assistants cannot send arbitrary data over AVS to a skill because AVS restricts messages to include lower-case letters and numbers only. This creates a challenge for sending ciphertext to a skill. To address this challenge, we encode the ciphertext using RFC 4648's base32 encoding that converts 5-bit data chunks to a code comprised of upper-case letters and the numbers between 2 and 7. With base32 encoding, each 5-bytes of ciphertext is converted to 8 characters, padded with '=' characters in case the encoded message's length is not a multiple of 8. Finally, we convert upper-case letters in base32 to lower-case and remove all the trailing '='. Decoding is done in a similar manner.

Skill support. A third-party skill can offer this functionality by developing a skill-specific companion extension. This skill and its companion extension only communicate through AVS. The extension sends the encryption key and encrypted messages to the skill in the same way as regular messages (i.e., using AVS). Consequently, the skill must register specific *intents*, *sample utterances*, and *slots* with AVS to let the extension achieve this goal. An intent represents an action that fulfills a user's request. The sample utterances indicate the pattern of the words users can say to invoke intents. And slots are the optional arguments of intents.

Slots are defined with different types. Amazon provides several built-in slot types to capture first names, phone numbers, city names, etc. In addition to the built-in slot types, users can leverage a specific slot type for capturing users' generic queries. This slot type is `AMAZON.SearchQuery`, which is designed to be used in search engine skills or any skill that needs to capture complete phrases from users. We use this slot type to receive the key from the assistant.

In the case of the secure conversation extension, the skill should register the following intents with AVS:

- Intent1: KeyIntent

  Sample utterance: key is {KEY}

  Slots: KEY ; Slot type: AMAZON.SearchQuery

- Intent2: SearchIntent

  Sample utterance: Search for {PHRASE}

  Slots: PHRASE ; Slot type: AMAZON.SearchQuery

This raises a challenge. To protect user's privacy, AVS does not allow the first intent that launches the skill to contain a slot with `AMAZON.SearchQuery` type. This limitation prevents us from converting the user's first request to a phrase such as *Alexa, open secret health with the {KEY}*. Therefore, we share the encrypted symmetric key at the second request to the skill, as described earlier in the workflow. Besides, Amazon also enforces another limitation on the usage of a slot type used in this skill (i.e., AMAZON.SearchQuery) for privacy purposes. Based on Amazon's rules, any `AMAZON.SearchQuery` slot should be accompanied by carrier phrases and cannot be used alone in an utterance [68]. This limitation is why we added the phrase "search for" to the beginning of the utterances of our SearchIntent intent.

4.6.2 Anonymous Query

This extension lets a user issue sensitive queries anonymously. The query is relayed indirectly through a *mixer skill* that prevents AVS or any other skill (including the mixer itself) from correlating the user's identity with the query's content.

This extension forms a secure channel with the mixer. Upon receiving messages from a user, the skill cannot identify the user because AVS never shares any of the user's identity

information with third-party skills. Thus, the mixer skill receives the user's query but does not know the user's identity. On the other hand, AVS knows the user's identity but does not have access to the query's contents. This separation ensures that the sensitive query remains anonymous. Note that we assume that AVS and the mixer skill do not collude.

Workflow. First, the user must invoke the mixer skill explicitly by saying "Alexa, Open Query Mixer". The mixer skill asks the user to submit the query to be anonymized, and the extension sends the query to the mixer over the secure channel. The mixer skill decrypts the query and submits it in plaintext format to AVS. Upon receiving the response, the skill forwards it back to the user over the secure channel.

We implement a prototype mixer for demonstration purposes. Our mixer reorders the requests, adds randomized latency to each request, and submits them to AVS. Moreover, we assume that the mixer skill has access to a small network of Alexa-enabled devices to submit the queries.

4.7 Implementation

We implement MegaMind on top of Amazon's Alexa Voice Service SDK. Therefore, MegaMind is compatible with all built-in and third-party skills deployed on the Alexa ecosystem.

4.7.1 Key Implementation Components

MegaMind engine. The MegaMind engine is the conductor orchestrating all other components. The Alexa SDK sends an IPC signal to the MegaMind engine whenever it detects the wake word. It then waits until it receives the processed user's command from the MegaMind engine using another IPC channel. Upon receiving the wake-word detection

signal, the MegaMind engine uses the speech-to-text engine to obtain the transcribed text of the user's command. MegaMind engine similarly processes the response from AVS. Besides, it forwards the altered response to a text-to-speech engine to read it for the user.

Runtime sandbox. We use the Firejail sandbox [56], which uses Linux namespaces and seccomp, to completely isolate the execution of the action function from the rest of the system.

Using the sandbox, we enforce the following restrictions. First, we limit file system accesses. We configure Firejail to only allow access to the random number generator, libraries, and binary programs that are essential for execution of action functions (which are written in Python) including Python packages on cryptography and natural language processing. Moreover, for performance purposes, we allocate a temporary subfolder in the RAM-based `/tmp` directory to be used as a home directory for the sandbox, which is needed for temporary storage and communication with MegaMind. Second, we disallow sandboxes to communicate with the outside world by disabling network access. Third, using seccomp filters, we limit the syscalls that can be executed. We only allow `open`, `read`, and `write` syscalls. Finally, we configure Firejail to limit the resources such as memory, number of files and size of files, and the CPU time available to a sandbox.

Speech-to-Text conversion. We use Mozilla DeepSpeech as our speech-to-text engine for the conversion. We choose DeepSpeech because it outperforms all of the open source speech-to-text conversion implementations that we have tested, including Kaldi, CMUSphinx, Julius, and Simon. According to Mozilla, DeepSpeech achieves a 7.5% word error rate on LibriSpeech clean database, and can convert speech to text faster than real-time even on a single core of a RPi 4 board [39]. Moreover, our engine uses the Voice Activity Detection (VAD) algorithm to detect the end of utterances and responses based on the silence detection.

Text-to-speech conversion. For our text-to-speech conversion, we use pico2wave from SVOX [66]. pico2wave is fast; it converts the text to speech in less than 1 ms on a normal laptop. The output voice of pico2wave sounds less artificial than its other alternatives such as eSpeak engine on Linux.

NLP helper functions. We implement MegaMind NLP helper functions using Python natural language processing toolkit (NLTK). MegaMind helper functions lie within three main categories.

The first category includes helper functions that search words in a database. Examples are functions that look for first/last names and profanity. For the former, we use a database including the top 5000 of all the first/last names registered for a Social Security card in the United States since 1879 [22]. For the latter, we use a list of 1383 profane words in English from CMU [64]. The second category includes helper functions that look for a predefined structure in the sentence. Examples are helper functions that find phone numbers or U.S. addresses in sentences. The third category includes helper functions that use information about a word's meaning. Examples are helper functions that find synonyms and similar words in the sentences, or functions that find a specific type of content such as violent content or adult content. We implement these helper functions with the help of NLTK and the WordNet database [72].

4.7.2 Performance Optimizations

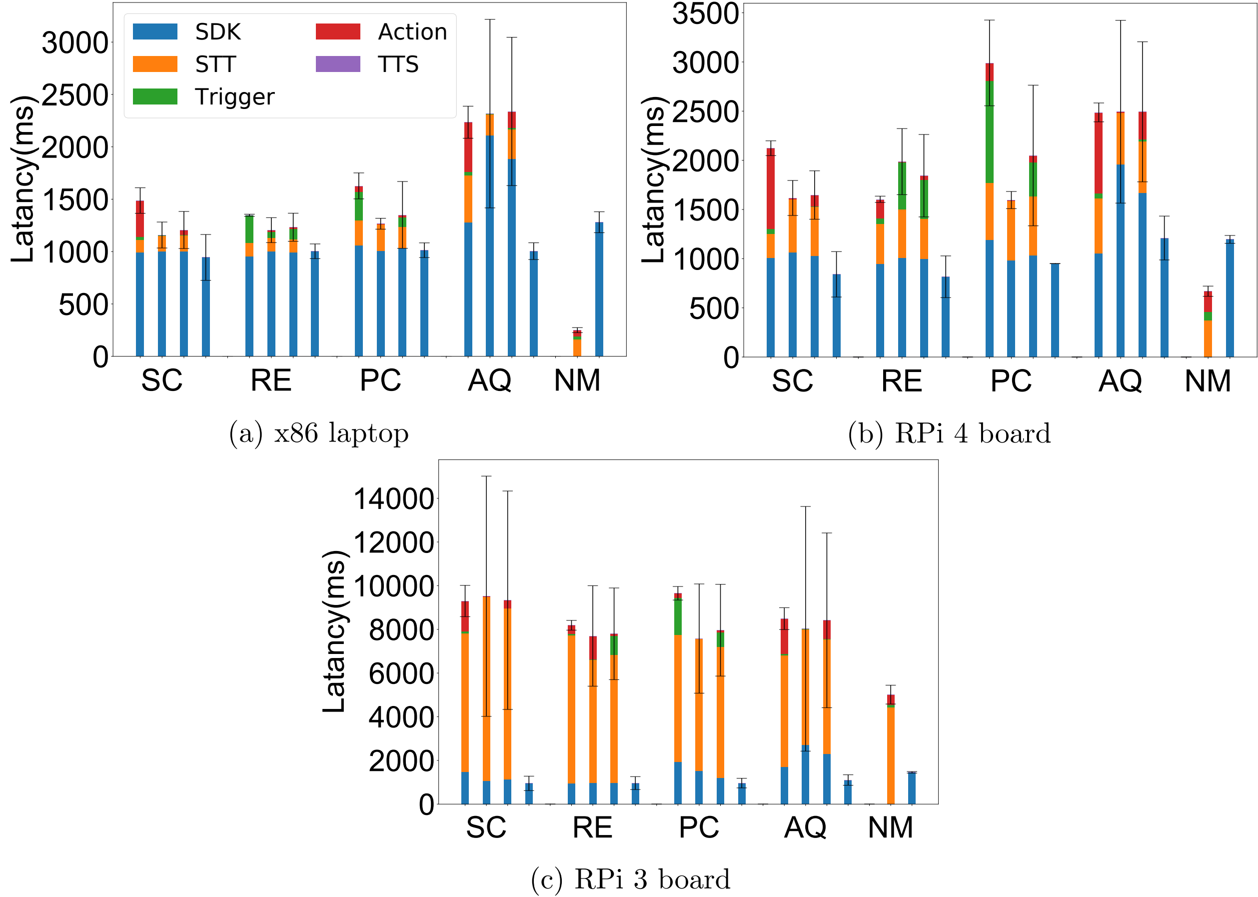
Sandbox pool. One source of overhead in our earlier prototypes was the sandbox initialization time. To avoid this high latency, we use a *sandbox pool*, which MegaMind initializes at boot time. This optimization reduces the latency by 420 ± 50 ms on a laptop, and 295 ± 20 ms on a RPi 4 board.

Text submission to AVS. Another source of overhead in our earlier prototypes was the time needed to covert the modified user utterance to audio in order to submit to AVS. To eliminate this overhead, we used another API of AVS that allows submission of requests in text format. This API is used in Alexa Developer Console for testing third-party skills. Moreover, the response from AVS, which is normally in audio format, includes the corresponding text of the response as well, when we submit the requests in text format. This eliminates the need for converting the response to text before passing it to the extensions, further reducing the latency. This optimization reduces the latency by about 310 ± 20 ms on a laptop and 630 ± 90 ms on a RPi 4.

Stream processing. If we wait until the end of the utterance and then start converting the recorded audio to text, it adds several seconds of latency. Therefore, we use stream processing for the conversion. In its recent versions (> 0.6), DeepSpeech provides a full streaming API. We use webRTC's VAD to collect 300 ms of audio from the microphone and pass it to the DeepSpeech streaming API. This way, regardless of the utterance length, we can have the text after around 300 ms. This optimization reduces the latency by 2000 ± 1500 ms.

4.8 Evaluation

We evaluate the performance and effectiveness of MegaMind. We deploy MegaMind on three platforms, a laptop, a RPi 4 board, and a RPi 3 board. On the laptop, we use a VMware virtual machine with 4 CPU cores and 2 GB of RAM. The laptop uses a 2.6 GHz Intel Core i7 x86 CPU with 4 cores, 8 GB of RAM, and Intel hardware virtualization (VT-x). RPi 4 uses a 1.5 GHz ARM Cortex-72 with 4 CPU cores and has 2GB of RAM. Finally, RPi 3 uses a 1.2 GHz ARM Cortex-53 with 4 CPU cores and has 1 GB of RAM. Our ARM prototypes represent lower-end mobile devices such as smartphones, modestly-powered



(a) x86 laptop

(b) RPi 4 board

(c) RPi 3 board

Figure 4.4: *Latency breakdown for different extensions for three platforms. For each extension, first, second and third bars, respectively, show the average latency for first, middle, and overall commands in a session. The last bar shows the baseline latency for that extension.*

standalone assistants, and embedded one. Our x86 prototype, on the other hand, represents higher-end and more powerful assistants.

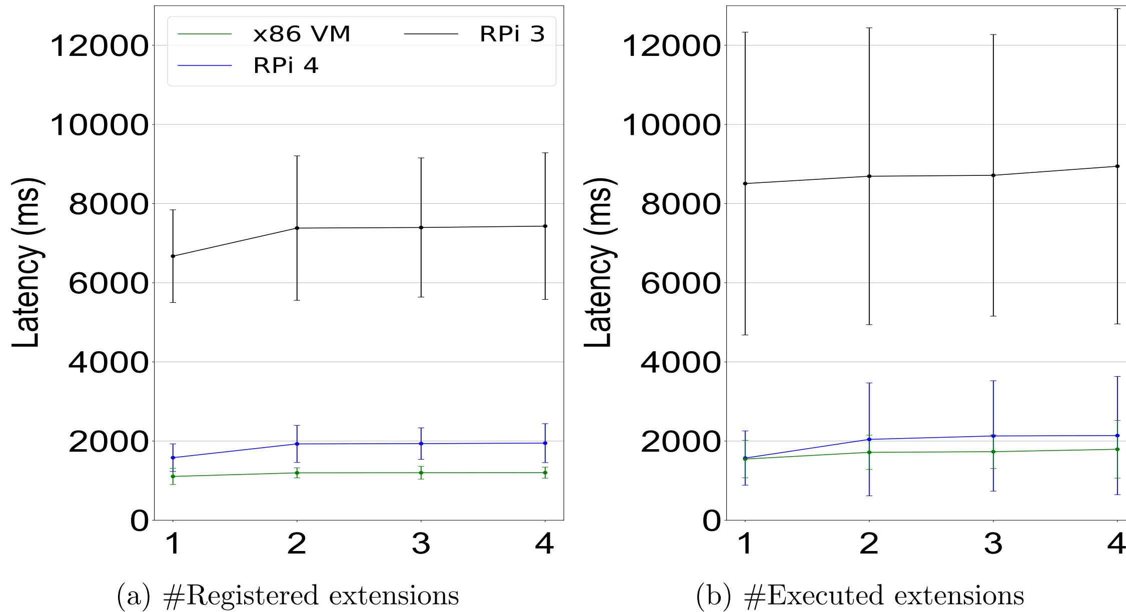
4.8.1 Performance

Conversation latency. The latency of responses is one of the critical factors in user's satisfaction with a voice assistant. We measure the latency for several extensions and report them in Figure 4.4. The figure shows the breakdown of the latency, showing contributions from local speech-to-text conversion, NLP helper function evaluation, and execution of computationally-heavy action functions. In addition, the figure shows the latency for the

first utterance in a session, *all utterances after the first one*, and *all utterances including the first one*. We show the results as such since the first utterance in a session suffers from higher latency than the other utterances in the same session, for several reasons. First, the action function execution initializes at this utterance. Second, extensions, specifically companion extensions, perform heavy computations at the first utterance. The latency also depends on the length of an utterance. As a result, we test each extension with a session consisting of five utterances with different lengths. To further reduce the measurements' noise, we repeat each session five times. The final latency for each extension is the average latency of twenty-five measurements in five different sessions.

As the figure shows, different types of extensions have different latency profiles. First, secure conversation (SC) and Anonymous query (AQ), have heavy initialization and impose higher latency on the first utterance. As we can see in the figure, most of this latency comes from the execution of the action function. Second, for redaction (RE) and parental control (PC) extensions, evaluating the trigger functions imposes the highest latency. This is because of the usage of NLP helper functions in the trigger rules of these two extensions. Finally, night mode (NM) and skill limiter (SL) experience a small latency since neither their action functions nor their rule evaluations are computationally intensive. Please note since these two extensions discard the utterances before submission to the SDK, there is no reported SDK latency for them in the figure.

Speech-to-text conversion, on average, imposes similar latency to each extension in each platform. On the laptop and RPi 4, speech-to-text performs near real-time and imposes less than 400 ms of latency on average. However, for the weaker platform, RPi 3, speech-to-text conversion imposes notable latency. This explains poor performance results on this board. As MegaMind relies heavily on local computation, it requires adequate compute power on the voice assistant. However, RPi 3 has much less computation power compared to RPi 4 [42]. Fortunately, our results show that a device as inexpensive as RPi 4 can



(a) #Registered extensions (b) #Executed extensions

Figure 4.5: *Impact of number of extensions on latency.*

provide adequate compute power. Most of the latency on weaker devices is for speech-to-text conversion. Hence, such devices (e.g., smartwatches) can offload the speech-to-text conversion to another edge device like a smartphone and still be able to deploy MegaMind.

Impact of the number of extensions. We next evaluate the effect of the number of extensions on latency, in two steps. First, we evaluate the impact of *the number of active yet not triggered extensions*. In this experiment, we measure the average latency for a sequence of utterances that do not trigger any of extensions. This way, we measure the overhead of evaluation of the trigger rules for these extensions, but not their action functions. To increase the number of extensions, we add the following in order: (1) secure conversation, (2) redaction, (3) night mode, and (4) skill limiter. Figure 4.5a shows the results. It shows that increasing the number of enabled extensions does not have a notable impact on the overall latency. Only adding the redactor, which uses NLP helper functions slightly increases the overall latency of MegaMind.

Second, we evaluate the effect of *the number of triggered extensions*. This experiment captures not only the impact of evaluation of trigger rules, but also execution of actions func-

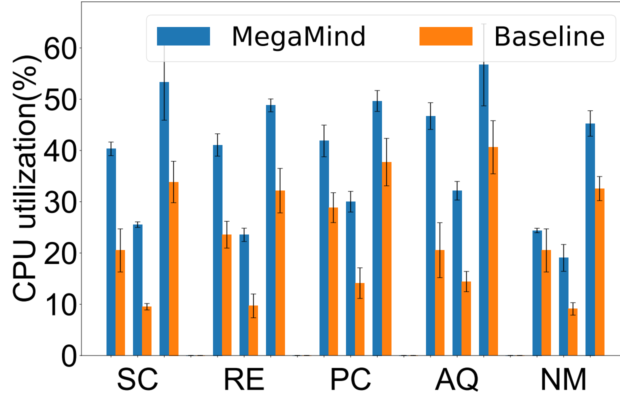


Figure 4.6: *Extensions CPU utilization. For each extension, first, second, and third bar groups, respectively, represent laptop, RPi 4, and RPi 3.*

tions. For this experiment, we use utterances that trigger all of the extensions. Please note that we modify the discarder's action function for this experiment to avoid discarding the utterances when they get triggered. Figure 4.5b shows the results. It shows that even if an utterance triggers four MegaMind extensions, the overall latency is only slightly higher than the latency of only one extension. This is because most of MegaMind latency comes from speech-to-text conversion (which executes once per command).

CPU utilization. We also measure the CPU utilization of each extension using the same experimental setup. Figure 4.6 shows the results. Since the unmodified SDK delegates almost all the computation to AVS, it is not surprising that running MegaMind increases the CPU utilization. For all platforms and all extensions, CPU remains idle most of the time. Thus, this increase in CPU utilization does not disrupt the normal execution of the assistant.

4.8.2 Effectiveness

MegaMind extensions can effectively provide security and privacy features for voice assistants. As mentioned in the introduction, we developed a few extensions and demonstrated their usage in a video demo. We have developed a simple banking skill that supports Mega-

Mind's secure communication alongside its MegaMind companion extension. In our experiments, the user securely communicates with this banking skill, logs in to his account, queries for his balance, and issues a transaction. We recorded the usage of this skill-extension pair and published it in our demo. In another instance, we showed the usage of an anonymous query extension. We showed how a user uses this extension to anonymously query for a medical condition.

Besides, we evaluate the impact of inaccuracies in speech-to-text conversion and NLP helper function components on the effectiveness of MegaMind. We note that these inaccuracies mainly impact sanitizers and discarders in MegaMind. They have, otherwise, minimal impact on companion extensions, such as secure conversation and anonymous query, for two reasons. First, companion extensions do not use NLP helpers in their trigger rules. Second, inaccuracies in the transcription can be easily mitigated by additional authentication methods employed by the companion skill.

We evaluate the effectiveness of MegaMind in four tasks: (1) detecting sessions, (2) redacting profanity, (3) redacting private information, and (4) preventing purchases. Table 4.3 summarizes the results. For each of the above tasks, we report results from two sets of experiment, one where we submit the test utterances in audio format hence requiring speech-to-text conversion, and one where we bypass the speech-to-text conversation and feed the accurate text of the utterances to MegaMind. These results help us understand the effectiveness of MegaMind in the presence of a highly accurate speech-to-text converter. Below, we discuss the effectiveness experiment results.

Effectiveness of skill ID detection.

To measure the accuracy of MegaMind in detecting explicit invocation of a skill, we test MegaMind with a combination of 100 standard built-in commands randomly chosen out of 190 Alexa built-in commands reported in [55] and twenty commands that we generate to

	Text		Voice	
Detection	FN	FP	FN	FP
New session	0%	8%	20%	8%
Profanity	0%	5%	10%	6%
Private info	0%	5%	15%	8%
Purchase	13%	1%	20%	2%

Table 4.3: *MegaMind's detection errors. FN stands for false negatives, and FP for false positives.*

ask Alexa to start a new session with a third-party skill. In generating these commands, we randomly chose the grammar to open the skill, and we chose skill names randomly from Alexa skill market.

Table 4.3 shows that MegaMind could find all of the commands aiming to start new session accurately. However, in a few cases MegaMind detects a false session start for a normal command. This is because the AVS grammar for starting a new skill has overlap with some of the Alexa's built-in commands. For example, a user can launch a third-party skill using the following grammar: *"[a request] from [skill invocation name]"* (e.g. *"Order pizza from great pizza shop".*) A built-in command such as *"Disconnect bedroom's echo device from John's phone"* follows the exact same grammar. MegaMind can potentially filter out all of these false new session detections by having a database of Alexa's published skills names.

Effectiveness of profanity redaction. For this experiment, we develop a custom skill, which tells jokes. We combine ten jokes containing profanity with one hundred clean jokes in a database, all randomly chosen from Laugh Factory [61]. Our result shows that MegaMind redacts all the profane words. However, since the database we used for bad words is conservative and contain dual-used words as well, MegaMind filters a few words in clean jokes as well.

Effectiveness of private information redaction. In this experiment, we mix twenty utterances containing private information such as first and last names, phone numbers, Social

Security Numbers, with 100 Alexa's standard commands randomly chosen out of 190 Alexa built-in commands reported in [55]. Our result shows that MegaMind could successfully redact all the private information in the utterances. However, in a few cases, MegaMind falsely redacts standard Alexa commands. These false alarms mostly happen in commands related to playing music, in which the redactor redacts the name of the artist. This problem only occurs when the redactor aims to redact all the matching first and last names. However, in real cases, a redactor can be configured only to redact the name of people using the device.

Effectiveness of purchase prevention. Out of 190 built-in Alexa commands reported in [55], 15 commands are listed as purchase-related commands. We measure how accurately MegaMind parental control extension can block these purchase-related commands. MegaMind parental control extension could find 13 of purchase-related commands using MegaMind NLP helper functions. However, two commands related to getting a taxi from ride-share skills were missed by MegaMind because there were no words associated with purchasing a product in these utterances. However, the parental control extension of MegaMind can easily block these utterances by disabling ride share skills.

Speech-to-text conversion accuracy. The word error rate for DeepSpeech speech-to-text engine is reported to be 7.5% [39]. However, this word error rate is for generic conversations. Voice commands may contain some words and phrases that were not present in DeepSpeech's training data-set. As a result, we use a database of Alexa built-in commands [55] consisting of 190 commands for Alexa to measure DeepSpeech's accuracy. We convert these commands to Speech using a human-like neural network-based cloud text-to-speech converter. We then convert back the spoken commands to text using DeepSpeech and measure the accuracy. Our experiments shows that DeepSpeech word error rate for this data set is 12.28%.

One other important aspect of speech-to-text conversion accuracy is in finding skill names. We measure the accuracy of MegaMind using DeepSpeech in accurately detecting the skill

names for 100 commands aiming to open 100 randomly selected skills from the top skills of Alexa skill market [44]. MegaMind could find the Skill names correctly in 82% of cases.

Speech-to-text conversion is a hot research topic and it is expected that the accuracy of local speech-to-text engines improves in the future. Our prototype uses a pre-trained English model for DeepSpeech, which has been trained with generic English speeches. However, we envision that in the near future, it will be possible to train a voice assistant-specific language model for DeepSpeech using voice assistant commands and skill names in order to further improve the accuracy. In addition, the DeepSpeech pre-trained models are only trained with noise-free pre-recorded standard and formal English speeches and do not support different accents and ambient noise. Training a robust language model requires a huge amount of labeled audio recording from users. Previously, only big companies had access to this kind of database. This task is getting feasible given the recent efforts from the open source community to build large transcribed databases of users' speeches by asking people to donate their voice to the database, and donate their time to validate the transcriptions [63]. For instance, Mozilla Common Voice project, at the time of writing this paper has reached 12000 hours of audio recording in 40 different languages, which 9500 hours of that is validated [63].

Chapter 5

Split-Trust Machine Model

Because of their ubiquity and portability, modern mobile systems such as smartphones are often used to run security-critical programs along with diverse, untrusted, and potentially malicious programs. For example, most of us perform routine financial tasks, such as banking and payments [181, 201], on our smartphones. And many of us run health-related programs, e.g., to receive test results and diagnoses from our health providers, and in some cases, to perform life-critical tasks, such as to control an insulin pump [230] or monitor breathing [188], on these same devices.

Realizing this computing paradigm should be straightforward. We can use an OS (or some other system software such as a hypervisor) to isolate these security-critical programs from other programs running on the same hardware. Yet, this has proven to be challenging in practice due to vulnerabilities in system software (e.g., OS, hypervisor, and device drivers) [237, 51, 29, 53, 54, 263, 198, 87, 108] and hardware (e.g., processor, memory, interconnects, and I/O devices including their firmware) [155, 173, 160, 235, 196, 241, 114]. Malicious programs that interact with the OS and use the same hardware can exploit these vulnerabilities to take control of the machine and any programs running on it. Therefore,

we must trust that the hardware and system software can effectively sandbox and neutralize malicious programs. This trust often proves to be unwarranted.

To address this challenge, a new approach has emerged. It uses *Trusted Execution Environments (TEEs)* to host security-critical programs without requiring trust in the OS. Unfortunately, today's TEEs still require us to trust the hardware and the security monitor implementing the TEE guarantees. This trust has also proven unjustified. Existing TEEs have fallen victim to various attacks, e.g., hardware-based side-channel attacks [91, 235, 98, 184, 183, 134, 212, 172, 265], attacks exploiting software vulnerabilities [95, 52, 204, 115], and attacks based on design flaws [141, 167, 245].

In this chapter, we present a solution to enable mobile systems to be used for both security-critical and non-critical programs. Our goal is to minimize both the number and the complexity of hardware and software components that need to be *strongly trusted* by the owner of the device in order to execute a security-critical program. As we will define in §5.1.1, we say that a component is strongly trusted if it needs to be able to withstand and neutralize adversarial inputs.

Our key principle is *provably exclusive access* to hardware and software components. That is, we design a solution to enable a security-critical program to *exclusively use complex hardware and software components and be able to verify the exclusive use.* Due to exclusive use, a component only needs to be *weakly trusted.* That is, it only needs to operate correctly in the absence of adversarial inputs.

More concretely, we present a hardware design for our computing system. Called a *split-trust machine model*, it comprises multiple trust domains, one or multiple for TEEs, one for each I/O device, one for a resource manager, and one for hosting a commodity OS and its programs. The trust domains are *statically-partitioned* and *physically-isolated*: they each have their own processor and memory (and one I/O device in the case of an I/O domain) and

do not share any underlying hardware components; they can only communicate by message passing over a hardware mailbox. Moreover, we introduce a few simple, *formally-verified* hardware components that enable a program to gain provably exclusive access to one or multiple domains.

We rigorously evaluate the required trust, i.e., the Trusted Computing Base (TCB), of this machine. We show that our machine significantly reduces the TCB compared to mainstream TEEs and achieves one close to the lower bound.

We present a prototype of our machine model on top of a a CPU-FPGA board (Xilinx Zynq UltraScale+ MPSoC ZCU102). We use the powerful ARM Cortex A53 CPU to host the commodity OS (PetaLinux) and its programs with high performance. We use the FPGA to build the other trust domains: two TEEs, a resource manager, and four I/O domains (an input domain, an output domain, a storage domain, and a network domain). We use (weak) microcontrollers for these other domains, including the TEEs. This choice as well as the small number of TEE domains is based on our observation that security-critical programs, unlike regular programs, are often not as computationally intensive, and the number of such programs that run simultaneously is typically small. In other respects, however, they are like normal programs: they start and stop, run in the background, do I/O, and so forth. Using our prototype, we show that the added hardware cost is small (i.e., 1-2%) compared to modern SoCs.

5.1 Background

5.1.1 Trust Definitions

The hardware and software components that need to be trusted for a program to execute securely form its TCB. In our work, we find that it is not adequate to determine whether a

component is trusted. We need to determine the type of trust.

More specifically, we define two types of trust: *strong trust* and *weak trust*. We say a component is strongly trusted if it needs to guard itself against *adversarial inputs*. For example, imagine an OS that is trusted to isolate a program from other malicious programs. The malicious programs can issue adversarial syscalls to the OS concurrently to the protected program. In such a case, the program owner needs to trust that the component (e.g., the OS) can prevent these other programs from exploiting any vulnerabilities (logical or implementation-related). Ensuring that a software or hardware component is not exploitable is very challenging, as demonstrated by the plethora of reported exploits. Therefore, *we believe that strong trust should be minimized for security-critical programs.* We note, however, that there are methods for hardening hardware and software components, such as formal verification. Strong trust is acceptable if a component is known to be adequately hardened against vulnerabilities.

We say that a component is weakly trusted if it just needs to operate correctly in the absence of adversarial inputs. For example, consider the same OS mentioned above, but assume that the security-critical program is the only one running on top of the OS (and assume no networking with the outside world). In such a case, the program owner only needs to trust that: (1) the component (e.g., the OS) does not exert buggy behavior under normal usage, i.e., when processing well-formed inputs, and (2) it is not compromised by an adversary before use and upon distribution (e.g., through implanted backdoors). These trust assumptions can be (more) easily met in practice by ensuring that: (1) component designers test it adequately under various expected usage models, (2) the source code of the component is available for inspection by security experts and users, and (3) users can verify the component before use through remote attestation. Therefore, *we believe that weak trust is acceptable for security-critical programs.*

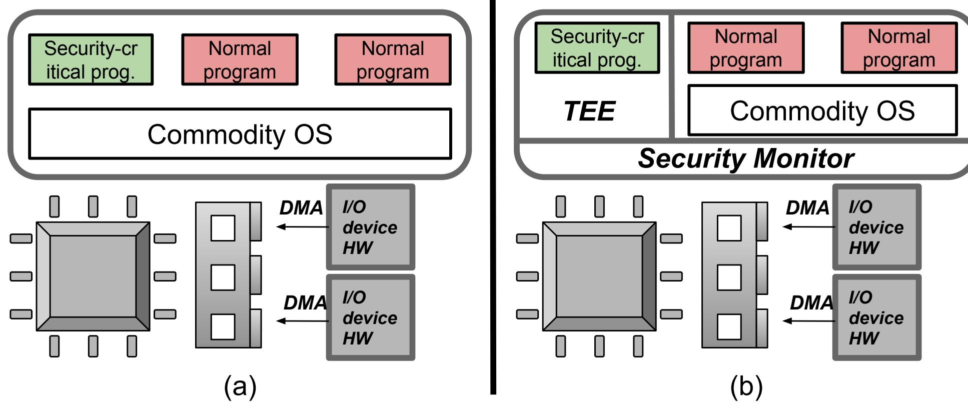


Figure 5.1: *(a) Traditional design where the OS isolates security-critical programs from normal programs. (b) Use of a TEE to isolate a security-critical program.*

5.1.2 Trust in Existing Systems

Historically, the OS has been a strongly-trusted part of the system. That is, the OS is trusted to isolate a program from other programs, benign or malicious, and provide three important security guarantees: integrity, confidentiality, and availability. Figure 5.1 (a) illustrates this traditional design.

As commodity OSes have become more complex over the years, more and more bugs and vulnerabilities have been found in them, allowing malware to exploit them and compromise the OS [51, 237, 53, 29, 263, 99, 198, 87, 108]. As an example, there have been about 1400 vulnerabilities reported in the Linux kernel just since 2016. This trend means that strong trust in the OS is no longer warranted.

There have been several attempts to build trustworthy OSes. These include microkernels [74, 171, 131, 159, 120], exokernels and library OSes [122, 150, 202, 89], formally verified OSes (and hypervisors) [159, 137, 138, 238, 190, 222, 168, 169, 227], and OSes written in safe languages [123, 145, 166, 189]. While effective, these solutions require replacing commodity OSes with a new OS. This is a challenging task due to the abundance of existing programs, device drivers, and developers for commodity OSes. More importantly, using these OSes still

requires strong trust in hardware, which is not warranted either, as we will discuss.

About two decades ago, a new approach started to gain popularity. The idea is to create an isolated environment, called a TEE, to host a security-critical program. This allows the use of a commodity OS, but relegates it to be only in charge of untrusted, normal programs such as games, utility apps, and entertainment platforms. The TEE enables a security-critical program to ensure its own integrity and confidentiality even if the OS is untrusted, but leaves the OS in charge of resource management (and hence the availability guarantee). Figure 5.1 (b) illustrates this design. It shows a *security monitor* is used to isolate a TEE from the OS. The security monitor can be implemented purely in software (i.e., a hypervisor) [102, 143] or using a combination of hardware and software. ARM TrustZone and Intel SGX are famous examples of the latter. Other examples include AMD Secure Encrypted Virtualization (SEV), Intel Trusted Domain Extensions (TDX), Apple's Secure Enclave Processor (SEP), ARMv9's Realms [135], and Keystone for RISC-V [163].

Despite their success, existing TEE solutions still require many components to be strongly trusted including the security monitor and several hardware components such as the very complex processor, memory, I/O devices in some cases, and dynamically-programmable protection hardware such as address space controllers and MMUs. Unfortunately, all of these components can be compromised by an adversary. For examples, hypervisors contain many vulnerabilities [54, 84]. The TEE OS in TrustZone also contains vulnerabilities and has been exploited in the past [95, 52, 204, 115]. AMD SEV has also been shown to contain several vulnerabilities due to design flaws, all of which have been exploited [141, 167, 245].

Hardware components have been exploited as well. Processor-based side-channel attacks have recently emerged as a serious threat to computing systems. For example, SGX enclaves and TrustZone have been compromised using several such attacks [91, 235, 98, 184, 183, 134, 212, 172, 265]. The core reason behind this is that existing solutions execute the untrusted OS and TEEs on the same hardware, forcing them to share the underlying microarchitectural

features such as caches [91, 172, 265, 183, 134, 212] and speculative execution engine [173, 235, 160, 98], as well as architectural ones such as virtual memory [184]. The memory subsystem has also proved vulnerable and fallen to Rowhammer attacks [155, 205, 236, 248, 136, 176]. The complexity of these hardware components ensures that many more such vulnerabilities will be discovered and exploited. For example, researchers have recently demonstrated a suite of new side channels using the interconnects [196], the x87 floating-point unit, and Advanced Vector extensions (AVX) instructions (among others) [241].

5.2 Key Goal and Principle

Key goal. Our goal in this work is to minimize the (1) number and (2) complexity of strongly-trusted components. The rationale for (1) is that it is difficult for hardware or software components to adequately protect themselves against adversarial inputs. The rationale for (2) is that it is easier to ensure that a component can fend off adversarial inputs if it is simple, which allows for comprehensive testing, analysis, and formal verification.

Key principle. Our key principle to achieve this goal is *provably exclusive access* to hardware and software components. That is, we design our machine to enable a security-critical program to *exclusively use complex hardware and software components and be able to verify the exclusive use*. More specifically, our goal is to have most components, especially complex ones such as the processor and system software, (1) be reset to a clean state before use, (2) then used exclusively by a security-critical program in a verifiable fashion through remote and/or local attestation, and (3) then again reset to a clean state right after use. In this case, such a component only needs to be weakly trusted as it does not need to worry about adversarial inputs while serving the security-critical program, nor does it need to worry about residual state from the security-critical program while serving other, potentially malicious, programs.

To realize this principle, we introduce a novel hardware design, the *split-trust machine model* (§5.3).

5.3 Split-Trust Machine Model

Modern machines leverage hardware with a *hierarchical privilege model*. That is, the hardware provides multiple privilege levels, each with more privilege than previous ones, with one all-powerful privilege level to "rule them all."[1] These privilege levels are implemented inside the CPU and use other programmable protection hardware components, such as MMUs. This model results inevitably in several complex, strongly-trusted components such as the processor, protection hardware, and system software, which if compromised, affect all programs.

In this chapter, we demonstrate a novel hardware design, the *split-trust* machine model, in which the hardware is split into multiple isolated trust domains. Each domain is intended for one aspect of the machine: one or multiple for TEEs, one for each I/O device (i.e., an I/O domain), one for a commodity OS and its untrusted programs (i.e., the untrusted domain), and one for a resource manager, which is in charge of *constrained* resource scheduling and access control. The benefit of the split-trust model is that a security-critical program can *exclusively* take control of and use its own domain and *exclusively* communicate with other domains, e.g., for I/O and IPC, hence significantly reducing the strongly-trusted components. (Exclusive inter-domain communication is enabled with a novel hardware mailbox abstraction that we will introduce in §5.3.2.) Figure 5.2 shows a simplified view of this hardware design. Next, we discuss its key aspects.

[1] A reference to Tolkien's The Lord's of the Rings.

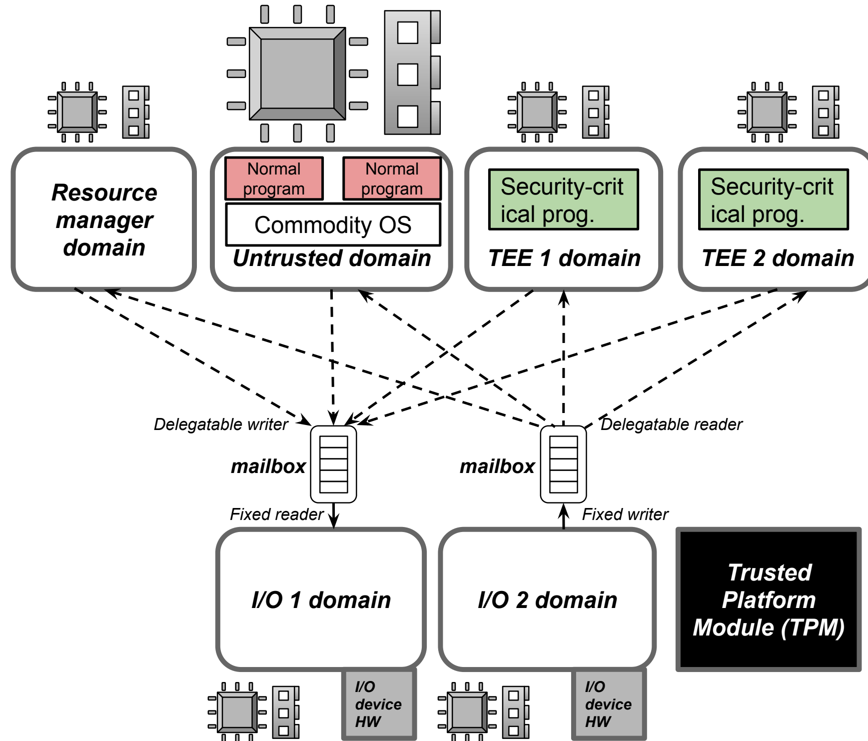


Figure 5.2: *Simplified overview of the split-trust machine model. The figure does not show all mailboxes for clarity.*

5.3.1 Static Partitioning and Physical Isolation

We follow two important principles in our hardware design: (1) domains must be physically isolated (i.e., share no hardware components), and (2) the isolation boundary between them cannot be programmatically and dynamically modified as *there is no trusted-by-all hardware or software component*. The latter implies that we cannot rely on programmable protection hardware, such as MMU, IOMMU, and address space controller, to enforce isolation. As a result, our design statically partitions the hardware resources between domains. Each domain owns its own hardware components (physical isolation) and that ownership is decided at hardware fabrication time and cannot be changed later (static partitioning).

More specifically, each trust domain has its own processor and memory. We use a powerful CPU for the untrusted domain, which accommodates a commodity OS and its (untrusted) programs, to achieve high performance. We use weaker microcontrollers for other domains

in order to keep the hardware cost small. Each domain has its own memory as well and domains do not (and cannot) share memory.

An I/O domain also has exclusive control of an I/O device, which is wired to and only programmable by the processor of that domain and its interrupts are routed to that processor. (We will discuss how DMA is handled in §5.3.5.)

5.3.2 Exclusive Inter-Domain Communication

To be able to act as one machine, the domains need to be able to communicate. We introduce a simple, yet powerful, hardware primitive for this purpose: *verifiably delegatable hardware mailbox* (mailbox for short).

At its core, a mailbox is a hardware queue, allowing two domains (i.e., the writer and reader) to communicate through message passing. A mailbox provides one-way communication. For two-way communication, two mailboxes are needed.

The key novelty of our mailbox is how it enables exclusive communication using its *delegation model*. A mailbox has a fixed end (reader or writer) and a delegatable one. The fixed end is hard-wired to a specific domain. The delegatable one is wired to multiple domains, but only one can use it at a time, enforced by a hardware multiplexer within the mailbox. This end is by default (i.e., after a mailbox reset) under the control of the resource manager domain. But the resource manager can delegate it to another domain, which is then able to *exclusively* communicate with the domain on the fixed end of the mailbox.

Figure 5.3 shows the design of the mailbox with a fixed reader. (The design of the fixed writer mailbox is similar.) For example, consider the serial output domain in our prototype. This domain is the fixed reader of a mailbox. Any domain with write access to the mailbox can (exclusively) send content to the output domain to be displayed in the terminal.

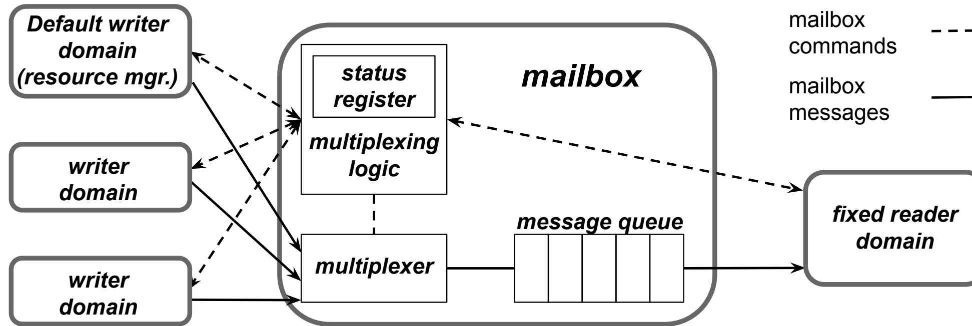


Figure 5.3: *Mailbox design.*

The delegation model of our mailbox has another important property: *limited yet irrevocable* delegation. When the resource manager delegates the mailbox to a domain, it sets a *quota* for the delegation in terms of both the maximum number of messages communicated and maximum delegation time. As as long as the quota has not expired (i.e., *a session*), the domain can use the mailbox and the resource manager cannot revoke its access to the mailbox. The session expires when either the message limit or the time limit expires. (The message limit can be set to infinite, but not the time domain, enforcing a temporal limit on the session length.)

This delegation model enables a limited form of availability, which we refer to as *session availability*. That is, a domain with exclusive communication access to another domain can be sure to retain its access for a known period of time or number of messages. This is critical for some security guarantees on personal computers. For example, a security-critical program can ensure that the User Interface (UI) will not be hijacked or covered with overlays when the program is interacting with the user [100, 256]. Or a security-critical program that has authenticated to and hence unlocked a sensitive actuator domain (e.g., insulin pump, §4.1) can ensure that no other program can hijack the session and manipulate the actuator.

As the resource manager is not trusted by other domains, the delegation must be *verifiable*. The mailbox hardware provides a facility for this verification. As Figure 5.3 shows, all domains connected to the mailbox can read a status register from the mailbox hardware.

The status register specifies the domain that can read/write to the mailbox and the remaining quota. The domain with delegated access can therefore verify its access and quota. (Other domains will receive a dummy value when reading the status register for confidentiality.)

5.3.3 Power Management

Our mailbox primitive cannot, on its own, guarantee session availability. This is because we need to ensure that during a session, the domains used by a security-critical program remain powered up (assuming there is adequate energy if battery-powered).

The Power Management Unit (PMU) normally takes commands from the resource manager. The resource manager uses this capability to reset other domains when needed, e.g., reset a TEE domain before running a new program, or apply Dynamic Voltage Frequency Scaling (DVFS) to manage the system's power consumption. (We do not support DVFS for the domains in our prototype. Hence, in the rest of the paper, we mainly focus on the reset interface, although similar principles can be applied to DVFS.)

However, the resource manager is not a trusted component; hence it may try to reset a domain during a session. Therefore, we add a simple hardware, called the *reset guard*, for the reset signals, which ensures that as long as the quota on a mailbox has not expired, the domains on both sides of the mailbox cannot be reset, hence ensuring session availability.

5.3.4 Hardware Root of Trust

A hardware root of trust is needed during remote attestation to convince the party in charge of a security-critical program of the authenticity of the hardware and the correctness of the loaded program. In a split-trust machine, we use a Trusted Platform Module (TPM) to realize the root of trust.

Why TPM? TPM, as specified by the Trusted Computing Group (TCG), is a tamper-resistant security co-processor connected to the main processor over a bus [229]. Hence, traditionally, it provides security features for the machine as a whole, such as the measurements of the loaded software. This makes TPM unsuitable for more fine-grained security features, such as remote attestation of a specific program. As a result, in-processor TEE solutions, such as SGX, integrate the root of trust in the processor itself, further bloating the strongly-trusted processor.

Our key insight is that TPM can provide fine-grained security features for a split-trust machine since different components of this OS run in separate domains. This allows the machine to enjoy the security benefits of TPM without suffering from its main limitation.

To integrate TPM into a split-trust machine, we need a different set of parameters from the ones found in existing TPM chips in order to provide one Platform Configuration Register (PCR) per domain and securely extend it with the measurement of software loaded in the domain. We do not provide more details here due to space limitations. We do, however, note that modifying the number of PCRs and their access permissions in TPM does not change its fundamental design principles. Indeed, the TPM specification does not specify these parameters [228], leaving them to implementers.

5.3.5 High Performance I/O

By default, the data plane of I/O domains are implemented over mailboxes. However, this raises a performance concern due to additional data copies (to and from mailbox). While the performance overhead is acceptable for TEE domains, it is not so for the untrusted domain. An important hardware primitive that enables a legacy machine to achieve high I/O performance is DMA. To safely use DMA in our machine, we introduce *domain-bound DMA*, defined with the following two restrictions:

- The DMA engine is hard-wired to only read/write to the memory of the untrusted domain.
- The DMA engine can stream data in/out of the I/O device only when the I/O domain is used by the untrusted domain.

When an I/O domain is used by a TEE domain, DMA is not used and data is transferred using mailboxes. But when the untrusted domain uses the I/O domain, data is transferred using domain-bound DMA for performance reasons. We achieve this with a simple hardware component called the *arbiter*, which is a switch that decides if the data streams of the I/O device is connected to a DMA engine or to a simple FIFO queue accessible to the I/O domain. As in a legacy machine, the untrusted domain may also use an IOMMU to further restrict DMA targets in order to isolate its own address spaces.

5.3.6 Domain and Mailbox Reset

As mentioned in §5.2, a key requirement for exclusive use of a domain is that the domain (and all its mailboxes) are reset to a clean state prior to and after use, in a way verifiable by the security-critical program. We reset the mailboxes directly in hardware upon delegation, yield, and session expiration. We leave the resetting of the domains to the resource manager, albeit under the limitations enforced by the reset guard (§5.3.3). Even though the resource manager is untrusted, this does not pose a problem since the program can verify, using local and remote attestation through TPM as well as some measures provided by the domain runtime that (1) a domain has been reset, (2) it has not been used since last reset, (3) it will be reset after use and before use by other domains.

5.4 Prototype

We have built a prototype of the split-trust hardware, on the Xilinx Zynq UltraScale+ MPSoC ZCU102 FPGA board. The SoC on the board has an FPGA as well as an ARM Cortex A53 processor. We use the ARM processor for the untrusted domain in order to achieve high performance for the commodity OS (PetaLinux) and its programs. We use the FPGA to synthesize 7 simple Microblaze microcontrollers (i.e., no MMU and no cache): two TEE domains, the resource manager domain, and four I/O domains (serial input, serial output, storage, and Gigabit Ethernet). For our TEE runtime, I/O services, and the resource manager, we leverage the Standalone library [249], which is single-threaded, to program the microcontrollers. We use the entirety of the main memory for the untrusted domain. For other domains, we use a total of 3.2 MB of on-chip memory including some ROM for bootloaders and some RAM. We run the TPM (emulator) [146] on a separate Raspberry Pi 4 board connected to the main board through serial ports. We use another Microblaze microcontroller to mediate the communications of the domains with the TPM.

In addition, we use the FPGA to synthesize the mailboxes (12 in total), the arbiter for DMA for the network domain (other domains do not support DMA), the reset guard, as well as 11 hardware queues for permanent domain connections (such as for all domains to communicate with TPM or for TEE domains to communicate with the resource manager). We note that we synthesize two types of mailboxes: *control-plane mailboxes* and *data-plane mailboxes*. The former has 4 messages of 64 B each and the latter has 4 messages of 512 B each. As a concrete example, our storage domain has 4 mailboxes: two for its control plane (send/receive) and two for its data plane (send/receive). Our mailbox interrupts a domain on every send and receive, which the domains can use to process incoming messages in a timely fashion and to ensure that outgoing messages have been successfully queued.

As mentioned in §5.3.1, an I/O device is only programmable by its domain. This includes

Property	Proved theorems
Exclusive access	Domains w/o exclusive access to mailbox cannot change which domain has exclusive access, nor the remaining quota.
	If a domain does not yield its exclusive access, its exclusive access is guaranteed as long as the quota has not expired.
	The domain with exclusive access to the mailbox can correctly read or write from/to the queue.
	The domains w/o exclusive access to the mailbox cannot read/write to the queue.
Limited delegation	When given exclusive access, a domain cannot use the mailbox more than its delegated quota.
	When the quota delegated to a domain expires, the domain loses exclusive access.
Exclusive Access verification	The domain with exclusive access can correctly verify its exclusive access and remaining quota.
	The domain on fixed end of mailbox can correctly verify the domain with exclusive access on the other end and the remaining quota.
Default exclusive access	After reset, the resource manager domain has exclusive access by default.
	The resource manager domain does not lose its exclusive access unless it delegates it.
	When a domain loses exclusive access (yield/expiration), the exclusive access will be given to the resource manager domain.
Confidentiality	Domains w/o excl. access cannot use mailbox's verification interface to find out which domain has excl. access and the remain. quota.
	Upon delegation/yield/expiration, the data in the queue is wiped.

Table 5.1: *Theorems we prove for our mailbox. Proving some of these theorems require proving multiple lemmas not listed here.*

access to registers and receiving interrupts from the I/O device. In our prototype, we use I/O interrupts only for the network domain and use polling for the rest. The interrupts to the network domain's microcontroller is from the FIFO queue that holds the packets and are only used when the domain serves a TEE domain (§5.3.5). When serving the untrusted domain, the domain-bound DMA engine directly interrupts the A53 processor on DMA completion.

When building the split-trust hardware, we faced numerous difficulties resulting from limitations imposed by the FPGA board. One limitation is noteworthy: the on-board SD card reader and flash memory are directly programmable by the A53 processor and hence could not be used for the storage domain. Our solution was to connect a MicroSD card reader directly to FPGA through Pmod [127]. This provides physical isolation for the storage domain, but significantly degrades its performance due to Pmod's limited throughput (§5.5.2).

5.4.1 Verified Hardware Design

The split-trust machine model has only four simple hardware components that are strongly trusted (see §4.3 for the TCB analysis): mailbox, DMA arbiter, reset guard, and ROM (for

bootloaders). We have implemented these components in 1630 lines of Verilog code as well as 800 lines of Python code to generate various mailboxes (i.e., mailboxes with different number of readers/writers) from a template design.

The simplicity of our strongly-trusted hardware components enables us to formally verify them. We use SymbiYosys to perform formal verification [132]. SymbiYosys is a front-end for Yosys-based formal hardware verification flows. We use the SMTBMC engine, which uses k-induction to formally verify safety features in hardware. Table 5.1 shows the list of theorems we prove for the mailbox (we omit the rest due to space limitation). Overall, we developed 3000 lines of SystemVerilog code for our hardware verification.

5.5 Evaluation

Our FPGA-based hardware implementation serves two purposes. First, we use it to estimate the hardware cost of our solution in terms of chip area. Second, it provides a bound on the performance impact of the solution. A deployed solution would likely replace the FPGA components with higher-performance non-reprogrammable ASIC elements, such as an integrated SoC or specialized chiplets [187].

5.5.1 Hardware Cost

We calculate an estimate for the number of transistors needed for our additional hardware components (all the components synthesized on the FPGA in our prototype). We calculate this estimate by measuring the number of look-up tables, flip flops, and block RAMs used by our hardware and converting them to transistor count using the following estimates: 6 NAND gates per look-up table [203], 6 transistors per NAND gate [226], 24 transistors for each flip flop [219], and 6 transistor for each bit of on-chip memory (assuming a conventional

FPGA resource	Count	Equivalent transistor count
Look-up table	63,289	2,278,404
Flip flop	57,033	1,368,792
Block RAM	26,840,190 (bits)	161,041,140

Table 5.2: *Cost of additional hardware in our machine.*

6-transistor SRAM cell [83]). Our calculation shows that our machine requires about 164.7 M additional transistors (161 M of which are used for on-chip memory). Table 5.2 shows the breakdown. This compares favorably with the number of transistors used in modern SoCs. For example, Apple A15 Bionic and HiSilicon Kirin 9000 use 15 B transistors [216, 128]. This means that, if our solution is added to an SoC or implemented as a chiplet, the additional hardware cost would likely be 1-2%.

5.5.2 Performance

We measure various performance aspects of our machine. Note that all domains except the untrusted one use an FPGA with a 100 MHz clock (the Ethernet controller IP uses an external 156.25 MHz clock). We repeat each experiment 5 times and report the average and standard deviation.

Mailbox performance. We measure the throughput and latency of communication over our mailbox. For throughput, we measure the time to send 10,000 messages of 512 B over our data-plane mailbox. For latency, we measure the round trip time to send a 64 B message and receive an acknowledgment over our control-plane mailboxes. We perform these experiments in two configurations: one for communication between the hard-wired ARM Cortex A53 (the untrusted domain) and an FPGA-based Microblaze microcontroller, and one for communication between two FGPA-based Microblaze microcontrollers. Table 5.3 shows the results. One might wonder why the A53-Microblaze configuration achieves lower performance. We believe this is because this configuration requires the data to pass the FPGA boundary, hence passing through voltage level shifters and isolation blocks [250].

Configuration	Throughput (MB/s)	Latency (μs)
A53-Microblaze	7.07±0	18.2±0
Microblaze-Microblaze	9.64±0.01	15.26±0.05

Table 5.3: *Mailbox performance.*

Configuration	Read throughput(MB/s)	Write throughput(MB/s)
Best-case	0.24±0.00	0.12±0.02
Untrusted dom.	0.24±0.09	0.09±0.02
TEE domain	0.21±0.00	0.11±0.01

Table 5.4: *Storage performance.*

Moreover, the FPGA is in a different clock domain than A53.

Storage performance. We measure the performance of our storage domain, which uses the mailbox for its data plane (i.e., no DMA). As mentioned in §5.4, our prototype uses a Pmod-based microSD card for storage. However, the Pmod connection limits the throughput of our storage service.

To measure the storage performance, we perform 2000 reads/writes of 512 B each. We evaluate three configurations: (1) a best-case performance where the storage domain directly performs the reads/writes (hence mainly stressing the Pmod-based microSD card), and (2) the untrusted domain as well as (3) a TEE domain storage performance (where the untrusted domain or a TEE domain uses the storage service). Table 5.4 shows the results. They show that our storage performance is mainly limited by the Pmod connection.

Network performance. We measure the performance of our network domain, which uses domain-bound DMA for high performance for the untrusted domain (§5.3.5). We evaluate three configurations: (1) a baseline where the A53 processor uses the Ethernet device (using an IP on the FPGA provided by Xilinx) and (2) the untrusted and (3) TEE domains using our network service. For measuring the throughput for the baseline and the untrusted configurations, we use iPerf; for round-trip time (RTT) measurements, we use Ping. For the TEE configuration, we develop custom programs for the measurements. For all experiments, we connect the board to a PC and measure the numbers reported by the measurement programs on the board. Table 5.5 shows the results. They show that our domain-bound

Configuration	Throughput (Mbit/s)	RTT (ms)
Baseline	943±0	0.17±0.01
Untrusted domain	943±0	0.17±0.02
TEE domain	0.022±0.001	23.92±0.02

Table 5.5: *Network performance.*

DMA is capable of matching the performance of a legacy machine.

Chapter 6

Related Work

6.1 Vulnerability Discovery

6.1.1 Remote I/O Access

Charm is a form of remote I/O. It enables software running in one machine to interact with an I/O device in another machine over a network connection. Hence, Charm is related to all systems that use remote I/O. The closest work to Charm is Avatar [259], a solution for dynamic analysis of binary firmware in embedded devices, such as a hard disk bootloader, a wireless sensor node, and a mobile phone baseband chip. Since performing analysis in embedded devices is difficult, Avatar executes the firmware in a virtual machine and forwards the low-level memory accesses (including I/O operation) to the embedded device. The remote boundary in Avatar is similar to the boundary used in Charm. However, Avatar focuses on a very different software and hardware. More specifically, it focuses on binary firmware of embedded devices whereas Charm focuses on open source device drivers of mobile systems. Moreover, in Avatar, the connections to the embedded device are low-bandwidth

UART or JTAG interfaces whereas Charm uses a USB interface. This, in turn, results in different underlying techniques used in the two systems. First, in its full separation mode, Avatar forwards all memory accesses to the embedded device, unlike Charm that ports the device driver fully to the virtual machine and only forwards I/O accesses. This results in poor performance in Avatar unlike Charm, which achieves performance on par with that of native mobile execution. To optimize, Avatar uses heuristics to perform some memory access locally. It also executes some or all of the firmware code directly on the embedded device. In contrast, Charm runs all the device driver code in the virtual machine. And for performance optimizations, it devises a custom low-latency USB channel and leverages the native execution speed of x86 processors. Avatar is limited to analysis of embedded firmware whereas our proposed solutions target analysis of device drivers in commodity mobile systems. These technical differences also make these solutions useful for different analysis techniques. For example, Charm can fuzz the device driver fully in a virtual machine.

Other forms of remote I/O exists for mobile systems as well, such as Rio [79] and M+ [195]. The main difference between Charm and these systems is the boundary at which I/O operations are remoted. Rio uses the device file boundary and M+ uses the Android binder IPC boundary. In contrast, Charm uses the low-level software-hardware boundary. Therefore, Charm uniquely enables the remote execution of the device driver. In both Rio and M+, the device driver remains in the machine containing the I/O device.

6.1.2 Analysis of System Software

Over the years, many static and dynamic analysis solutions have been invented for a wide range of applications such as safety, reliability, and security. In recent years, popular analysis techniques include taint tracking [191, 110, 257, 121], symbolic and concolic execution [92, 107, 96, 116, 252], unpacking and reverse engineering [156, 253, 152, 267], malware

sandboxing [3, 246, 90], and fuzzing [133, 97, 247, 240].

System-wide analysis. Many of these analysis frameworks are built on top of the virtualization technology and can support full-system analysis, including the low-level code such as kernel and device drivers [191, 110, 109, 257, 254]. For instance, Panorama [257] and DroidScope [254] can analyze the entire Windows and Android operating systems, respectively. Aftersight [109] uses virtual machine replay to feed recorded logs from a production system to a testing system in real time where more expensive analysis is run. kAFL is a hardware-based feedback-driven kernel fuzzer [211]. It uses the Intel Processor Tracer (PT) to collect execution traces in the hypervisor and use that to guide the fuzzer. Digtool is a kernel vulnerability detection solution based on a customized hypervisor, which can monitor various events in the kernel such as memory allocation and thread scheduling. Keil et al. fuzz wireless device drivers in a QEMU virtual machine [154]. To enable the driver to run in a virtual machine, they emulate the wireless interface hardware in software. Dovgalyuk et al. perform reverse debugging of device drivers in a QEMU virtual machine. They use GDB as well as record-and-replay in their debugging. Unfortunately, none of these solutions can be applied to device drivers of mobile systems. They can only support system software running within a virtual machine, e.g., device drivers for emulated and virtualized I/O devices (including direct device assignment for PCI-based I/O devices). Charm addresses this problem and is complementary to all of these solutions. In other words, Charm enables all of these dynamic analysis solutions to be applied to device drivers of mobile systems as well. Fuzzing is an effective dynamic analysis technique, which can be applied to the OS kernel and device drivers as well. Peach Fuzzer fuzzes the device drivers by running a fuzzer in a separate physical machine than the one with the I/O device [27]. While superior to running the fuzzer and driver in the same machine, their approach suffers from similar challenges that Syzkaller suffers from when fuzzing a mobile system directly (§2.1.3). Charm solves these problems by making it possible to run the device driver in a virtual machine.

In [180], Mendonça et al. fuzz the Wi-Fi interface card driver. They perform the fuzzing directly on a Windows Mobile Phone. In contrast, Charm enables the fuzzing to be performed in a virtual machine in a workstation, providing significant usability benefits.

DIFUZE automatically generates templates for fuzzing the kernel device drivers directly on mobile systems [111]. OctopOS is orthogonal and can benefit from DIFUZE for template generation.

Device driver or firmware analysis. The diversity of device drivers and their direct interactions with physical I/O devices create challenges for dynamic analysis. Static analysis, therefore, has been extensively used on device drivers [108, 87, 198]. Examples are symbolic execution solutions such as in SymDrive [206], S2E [106], and DDT [162], which can effectively analyze device drivers, and taint and pointer analyses as in DR. CHECKER [178]. Static analysis has the benefit of eliminating the need for the presence of actual devices. However, static analysis tools cannot uncover all the bugs and vulnerabilities in the drivers. They can only detect those for which the analyzer explicitly checks for. Moreover, static analysis solutions often suffer from large false positive rates due to imprecision.

Analysis of firmware running inside embedded devices faces similar challenges stemming from diversity as analysis of device drivers. Both static analysis [113] and dynamic analysis [259, 220] solutions have been used for firmware analysis as well. In contrast to this line of work, Charm focuses on modern mobile systems.

6.1.3 Mobile Testing

Several mobile testing frameworks have recently emerged. BareDroid analyzes Android apps directly on mobile systems [186]. SPOKE analyzes the access control policies of Android by running test cases directly on mobile systems [239]. The main motivation behind this line

of work is the fact that the system software of mobile systems are so different from that of x86 machines that these tests cannot be simply performed on existing virtual machines. Our motivation is in line with these systems as well. However, directly analyzing the device drivers in mobile systems is challenging, as we extensively discussed in the paper. Therefore, we enable these device driver to execute in a virtual machine for enhanced analysis.

6.2 Vulnerability Mitigation

6.2.1 Bug workarounds.

Talos inserts SWRRs (Security Workaround for Rapid Responses) into functions of an application in order to prevent the execution of known vulnerabilities [16]. Talos can be used to protect a vulnerable application until a patch is available. Talos performs static analysis to extract the right error number for a function and return that instead of executing the vulnerable function. In contrast, bowknots do not disable a function. They only undo a specific execution path that triggers a bug. In §3.6, we comprehensively compared bowknots with Talos.

6.2.2 Automatic fault recovery.

FGFT provides fine-grained recovery for faults in device drivers [151]. To do so, it checkpoints the memory and I/O device state on select entry points and restores them when a fault is detected. FGFT's key technique is to checkpoint and restore device state using existing power management code in device drivers. There are two important limitations that make this solution unsuitable to be used as a generic kernel bug workaround solution. First, checkpointing the state of an I/O device using power management facilities is not feasible for

all I/O devices. In fact, some of the devices that we tested in our evaluation (e.g., the camera of Nexus 5X smartphone) do not support this. Moreover, a checkpointing solution for the kernel memory is difficult to integrate into existing kernels. Virtual machine checkpointing exists; however, that does not apply to the kernels of real systems. Second, checkpointing the state of the system before every syscall is costly.

ASSURE uses rescue points for automatic recovery from faults in an application [221]. Rescue points are sites within an application that handle known errors. When faced with an unknown error, ASSURE restores the state of the application to a suitable and close rescue point, which then returns an error. However, ASSURE requires checkpointing the state at rescue points, which is expensive for syscalls and not feasible for all the hardware state.

Akeso uses recovery domains to undo a syscall or interrupt upon a fault [164]. Recovery domains log modifications to the kernel state and commit only upon successful execution. This allows the domains to undo the effects when facing a fault. Similar to hecaton, Akeso can undo a syscall that ends up in a bug trigger. However, Akeso's approach is not suitable for a bug workaround either. First, Akeso has significant performance overhead (1.08× to 5.6×). Second, Akeso does not support "code that write directly to external devices", which includes important parts of device drivers.

RCV automatically recovers from null pointer dereference and divide-by-zero errors [175]. It does so by handling the corresponding signals, repairs the execution by performing a default operation (e.g., return zero to a read from a zero address), monitors the effects of the repair in order to contain its effects within the application process, and detaches from the application when the effects are flushed. RCV is suitable for deployed applications as it helps them survive otherwise fail-stop errors. However, it does change the behavior of the application (even if slightly) and hence is not appropriate as a workaround solution.

6.2.3 Input filtering.

Another possible approach to work around a bug in the kernel is to filter those syscalls that trigger it. For example, VSEF uses execution-based filters to detect and then prevent exploits of a known vulnerability [192]. Sweeper monitors the execution of programs to detect attacks, analyzes the attack, deploys an antibody to prevent future exploits, and recovers the execution using the checkpoint/restore mechanism [232]. Vigilante generates a filter for preventing worms from exploiting vulnerable services [112]. However, there are important limitations for this approach to be used as a bug workaround. First, evaluating every syscall against a filter causes performance overhead. Second, discovering the exact condition and inputs under which a syscall triggers a bug is challenging. Third, there is currently no syscall filtering solution that can perform complex checks on the syscall input. Seccomp provides kernel syscall filtering but does not allow to maintain any state nor does it allow to check the arguments passed in memory.

6.2.4 Automated patching.

The goal of this line of work is to generate a correct patch for a bug automatically. Recent efforts do so by using simulated genetic processes to fix program faults [243], leveraging static analysis to patch race conditions [149], policing invariants to curb heap buffer overflows and control flow hijacks [200], utilizing the semantic analysis of test suites to correct program states [193], and using code annotations (contracts) to generate patch candidates [242]. In contrast, we focus on a workaround for a bug. Our goal is not to properly patch the bug, rather to provide a temporary solution until a patch is ready. Hence, our work is orthogonal to this line of work.

Hot-patching is a method for changing the behavior of binaries at runtime, commonly used for delivering patches without the need to reboot [225]. Linux kernel and kernel extensions

implement hot-patching by modifying the impacted functions and redirecting the execution flows [10] [9]. Recently, the urgent need for delivering security patches to fragmented Android devices has become a hot research topic. KARMA [104], VULMET [251], Instaguard [103], and Embroidery [266] extract rules and specifications from existing patches, and generate hot-patches for the fragmented Android kernel or user space binaries that are poorly maintained. These hot-patching mechanisms work assuming that the patches are available. In contrast, a workaround tries to mitigate a bug before a patch is available. Hence, our work is orthogonal to this line of work.

6.2.5 Error handling analysis.

Several efforts have attempted to identify defective error handlers. For example, CheQ [177] locates security checks and error handlers in the kernel by searching certain patterns, and leverages this information to catch unhandled errors and other bugs. APEx [153] identifies the error handlers based on the characteristics of error paths. EPEx [148] symbolically executes the test programs and explores error paths to find the mishandled exceptions. ErrDoc [231] leverages both symbolic execution and function pair matching to identify error handlers, and it automatically detects and then fixes incorrect or missing handlers. Hector [210] walks the control graph to identify the missing release statements in the error handlers based on a list of acquisition-release function pairs. EIO [139] and Rubio-González, et al. [209] present a method that uses data-flow analysis to detect unchecked errors as they propagate in the file system code.

hecaton identifies function pairs using a method similar to PF-Miner [174] and ErrDoc [231], which utilize string matching and path heuristics. However, there are two differences. First, PF-finder uses Longest Common Substring (LCS) as a metric as opposed to hecaton's string similarity score discussed in §3.4.1. Second, PF-finder discards the paired functions with the

exact same name, which can result in errors. For example, `regulator_set_voltage` function is used to both turn on and turn off a device.

6.2.6 Voice assistants security.

Our work on MegaMind is inspired by systems that provide security extensions for mobile operating systems, such as ASM [142] and ASF [85].

Almond [93] is an open source virtual assistant system. It uses a natural language interface and protects user's privacy by keeping their data locally. PrIA [147] is an intelligent assistant that provides personalized services, such as a news recommendation service, for the user without providing user's private information to cloud services. MegaMind shares Almond's and PrIA's visions of enhancing user's privacy when using assistants. Yet, we have designed MegaMind as a security and privacy extension to existing voice assistant systems, in contrast to these systems, which provide a new ecosystem or new services.

Use of assistants in a smart home setup creates security and privacy challenges when used by multiple people [261]. These challenges are different from those addressed by MegaMind, which focuses on security and privacy of using cloud services via voice assistants.

LipFuzzer [268] uses a linguistic-model-guided fuzzer to find semantic inconsistencies between the user and the voice assistant, resulting in the user talking to an unintended skill. While MegaMind's goal is orthogonal to LipFuzzer's, its extensions can alleviate some of these unintended results.

There is a line of work on enhancing the privacy of voice-based systems by eliminating personal features from audio recordings via local preprocessing [76, 75]. MegaMind's local speech-to-text conversion also eliminates all voice-based features. Although speech-to-text conversion requires more processing power, having the transcribed commands enables more

sophisticated language processing.

Other previous research has also identified the need for voice assistants to provide strong security defenses including authorization, access control, and privilege separation [117, 233, 124, 125, 223, 260]. Besides, there are previous empirical studies that highlight the importance of security and privacy of smart speaker applications [105, 170, 217].

There is a large body of work showing different classes of security attacks on voice assistants. *Inaudible voice attacks (IVA)* and *concealed voice attacks (CVA)* stealthily deliver voice commands to a voice assistant without the user knowing. BackDoor [207], Dolphin [262] and LipRead [208] use inaudible sounds transmitted over ultrasound frequencies to issue inaudible voice commands to virtual assistants. Similarly, research projects on concealed voice commands showed that devices continue to respond to wake words and utterances even when "mangled" to such a degree that they are unintelligible to users [234, 94]. Recently, CommanderSong [258], Lyexa [182], SurfingAttack [255], MetaMorph [101], and Abdullah et al. [73] proposed more elaborate and practical inaudible/concealed voice attacks.

Another important attack is the *voice squatting attack (VSA)*, where a malicious skill developer creates an invocation phrase similar to a legitimate skill, in the hope that sometimes the wrong skill may be invoked and data may leak [161, 264]. Another important attack is the *fake skill termination attack (FSTA)*, [264], where a malicious skill developer creates a long silent audio response in order to trick the user into thinking that no skill is running anymore, at which point the user may say something private.

6.3 Vulnerability Prevention

6.3.1 Security by physical isolation.

Using a separate, dedicated processor with its own memory and I/O devices for security-critical tasks is a recent hardware trend in personal computers. Apple has integrated the Secure Enclave Processor into its products [47] since about 2014 and used it to secure the user's secret data and to control biometric sensors (i.e., Touch ID and Face ID) [48]. Similarly, Google recently announced that Pixel 6 uses the tensor security core to host security-critical tasks such as key management and secure boot [158]. Our work takes the concept of security by physical isolation further by allowing programs (including those that rely on I/O devices) to use dedicated processors by developing a model for how that can be done safely.

Notary [82] safeguards approval transactions by running its agent on a separate SoC from the ones running the kernel and the communication stack. Our work shares the idea of using physically-isolated trust domains and also resets the domains before and after use by other programs (although we have not formally verified our bootloader code that cleans up the state upon reset, but plan to investigate adopting Notary's deterministic start). We show how to safely mediate access to shared I/O devices for a workload of concurrent security-critical and untrusted programs. Likewise, I/O-Devices-as-a-Service (IDaaS) suggests that I/O devices should have their own separate microcontrollers (and observes that they often do) and advocates for hardening their interfaces against potentially malicious kernel behavior [78]. Our approach also uses separate I/O microcontrollers but does not require strong trust in the microcontroller software, by resetting the I/O domain between uses.

6.3.2 Secure I/O for TEEs.

SGXIO uses a hypervisor and a TPM to create a trusted path for an SGX enclave to access an I/O device [244]. The solution requires the enclave program not only to trust SGX's firmware and hardware, but also the hypervisor. Cure [86] adds a few hardware primitives in order to allow the security monitor to assign a peripheral (i.e., access to MMIO registers and DMA target addresses) to an enclave. These primitives are designed to be programmed by a trusted-by-all security monitor (unlike our work).

Helios [194] leverages satellite kernels to expose a uniform set of abstractions to applications running on heterogeneous hardware. Moreover, I/O is accessed through remote message passing, which has similarities to our I/O services. Barrelfish [88] runs a separate kernel on each core in a multicore machine for better scalability and uses message passing for communication between kernels.

M^3 [81] and M^{3}x [80] enable the use of heterogeneous processing elements (PEs) by hiding them behind a hardware component, namely Data Transfer Unit (DTU). Since an application runs on a separate PE, it does enjoy physical isolation. However, in M^3, a kernel (running on its own PE) makes unconstrained access control decision and hence needs to be strongly trusted.

6.3.3 Time protection.

Ge et al. add time protection to seL4, which closes many of the available side channels in commodity processors [130]. As the paper mentions, some processors do not provide mechanisms needed to close channels. Moreover, channels using busses could not be closed, and they have recently been shown to be effectively exploitable [196]. Our approach of using completely separate hardware for security-critical programs addresses these concerns

for these programs. We do, however, note that our approach (as it stands) does not scale to support all (normal) programs, which may have their own security needs. Therefore, we believe that time protection remains an important abstraction to be explored for when the same processor is asked to host multiple programs.

6.3.4 Other TEE solutions.

Flicker [179] uses the late launch feature of Intel Trusted Execution Technology (TXT) [129], to exclusively run a program on the processor. The exclusive use of the hardware results in minimizing the strongly-trusted components. However, Flicker's design requires stopping all other programs (including untrusted ones) when running a security-critical program. Our approach can run untrusted programs and security-critical programs concurrently. Consider our secure insulin pump program (§4.1), which might need to be run frequently while the user is actively doing other, less security-critical, tasks on the main processor. Realizing this in Flicker can result in significant disruptions to other programs and to the user as a result.

Komodo is a verified security monitor that can create enclaves for security-critical programs [126]. Use of formal verification warrants the strong trust in the security monitor, but not the ARM processor that hosts both security-critical and untrusted programs. Sanctum uses hardware modifications to RISC-V alongside a software security monitor to create isolated enclaves. Compared to SGX, Sanctum enclaves are protected against both cache and page fault side-channel attacks. While this is important, Sanctum does not address other potential hardware vulnerabilities such as side channels through interconnects.

Chapter 7

Conclusions

In this dissertation, we presented four system solutions that improve security of mobile devices. We showed how these systems effectively contribute in fighting bugs/vulnerabilities through discovery, mitigation and prevention of bugs/vulnerabilities. We showed that our systems improve security of mobile devices with reasonable and minimal added cost, while they do not harm performance of mobile devices.

First, we presented *Charm* a system solution that improves vulnerability discovery through facilitating dynamic analysis of mobile device drivers. Charm enables application of various existing dynamic analysis solutions, e.g., interactive debugging, record-and-replay, and enhanced fuzzing to these device drivers. Our extensive evaluation showed that Charm is easy to use, achieves decent performance, and is effective in enabling a security analyst to find, study, and analyze driver vulnerabilities and even build exploits.

Second, we presented *bowknots* and showed how they can be used to workaround existing kernel bugs. We also presented *Hecaton* that can automatically generate bowknots for mobile kernels. Bowknots maintain the functionality of the system even when bugs are triggered, are applicable to many kernel bugs, do not cause noticeable performance overhead, and have

a small kernel footprint. Our evaluations show that bowknots are effective in mitigating bugs and security vulnerabilities and preserve the system functionality in most cases. We also show how bowknots improve vulnerability discovery by eliminating unnecessary reboots during kernel fuzzing.

Third, we presented *MegaMind*. We showed how it enables usage of novel and useful security and privacy extensions in voice assistant with minimal overhead. We demonstrated several such extensions, including one for secure communication with a third-party skill and one for issuing queries anonymously, both of which bring a level of unprecedented security for voice assistants. We presented a prototype that works with the existing Alexa Voice Service ecosystem and showed that it achieves a low conversation latency even on inexpensive hardware, such as a Raspberry Pi 4 board. We also showed that MegaMind is effective in achieving various security and privacy goals. We showed how MegaMind helps users to keep their existing voice assistant secure despite their vulnerable design.

And finally, we presented *split-trust machine model* that is more secure by design and let users run their security-critical applications on their personal mobile devices. We present a hardware design with multiple statically-partitioned, physically-isolated trust domains, coordinated using a few simple, formally-verified hardware components. We describe a complete prototype implemented on an FPGA and show that it incurs a small hardware cost.

Bibliography

[1] ANDROID FRAGMENTATION VISUALIZED (AUGUST 2015). http://opensignal.com/assets/pdf/reports/2015_08_fragmentation_report.pdf.

[2] Android Security Bulletins. https://source.android.com/security/bulletin/.

[3] Anubis: Analyzing Unknown Binaries. http://anubis.iseclab.org/.

[4] Code coverage tool for compiled programs (KCOV). https://github.com/SimonKagstrom/kcov.

[5] Google Issue Tracker: Issues. https://developers.google.com/issue-tracker/concepts/issues.

[6] Google Syzkaller: an unsupervised, coverage-guided Linux system call fuzzer. https://opensource.google.com/projects/syzkaller.

[7] Instruction for using the Syzkaller to fuzz an Android device. https://github.com/google/syzkaller/blob/master/docs/linux/setup_linux-host_android-device_arm64-kernel.md.

[8] iPerf - The ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr/.

[9] Livepatch. https://www.kernel.org/doc/Documentation/livepatch/livepatch.txt.

[10] Oracle Ksplice. https://ksplice.oracle.com.

[11] The Kernel Address Sanitizer (KASAN). https://github.com/google/kasan/wiki.

[12] The Kernel Memory Sanitizer (KMSAN). https://github.com/google/kmsan/blob/master/README.md.

[13] The Kernel Thread Sanitizer (KTSAN). https://github.com/google/ktsan/wiki.

[14] The Kernel Undefined Behavior Sanitizer (KUBSAN). https://www.kernel.org/doc/html/v4.11/dev-tools/ubsan.html.

[15] The ultimate, open-source X86 and X86-64 decoder-disassembler library. https://zydis.re/.

[16] What's New in Android Security (Google I/O '17) - Video. https://www.youtube.com/watch?v=C9_ytg6MUP0.

[17] USB Gadget API for Linux. https://www.kernel.org/doc/html/v4.13/driver-api/usb/gadget.html, 2004.

[18] american fuzzy lop. http://lcamtuf.coredump.cx/afl/README.txt, 2015.

[19] Android vs iPhone boot times tested: which one is the fastest? https://www.phonearena.com/news/Android-vs-iPhone-boot-times-tested-which-one-is-the-fastest_id69582, 2015.

[20] Qualcomm launches bug bounty program for Snapdragon chips, modems. https://www.zdnet.com/article/qualcomm-launches-hardware-bug-bounty-program/, 2016.

[21] Toddler asks Amazon's Alexa to play song but gets porn instead. https://nypost.com/2016/12/30/toddler-asks-amazons-alexa-to-play-song-but-gets-porn-instead/, 2016.

[22] US First Names Database . https://data.world/len/us-first-names-database, 2016.

[23] Building a Pixel kernel with KASAN+KCOV. https://source.android.com/devices/tech/debug/kasan-kcov, 2017.

[24] Cortana in the car: Microsoft launches new automotive tech platform, strikes Renault-Nissan partnership. https://www.geekwire.com/2017/cortana-car-microsoft-launches-new-automotive-tech-platform-strikes-renault-nissan-partnership/, 2017.

[25] Google Home can now power on your Vizio TV. https://www.cnet.com/news/google-home-can-now-power-on-your-vizio-tv/, 2017.

[26] How syzkaller works. https://github.com/google/syzkaller/blob/master/docs/internals.md, 2017.

[27] Peach Fuzzer for Driver Fuzzing Whitepaper. https://www.peach.tech/datasheets/driver-fuzzing/peach-fuzzer-driver-fuzzing-whitepaper/, 2017.

[28] Amazon Alexa-Powered Device Recorded and Shared User's Conversation Without Permission. https://www.wsj.com/articles/amazon-alexa-powered-device-recorded-and-shared-users-conversation-without-permission-1527203250, 2018.

[29] Bugs and Vulnerabilities Founds by Syzkaller in Linux Kernel. https://github.com/google/syzkaller/blob/master/docs/linux/found_bugs.md, 2018.

[30] Google Assistant – LIFX.com. https://www.lifx.com/products/google-assistant, 2018.

[31] Hey, I didn't order this dollhouse! 6 hilarious Alexa mishaps. https://www.digitaltrends.com/home/funny-accidental-amazon-alexa-ordering-stories//, 2018.

[32] Is Alexa Listening? Amazon Echo Sent Out Recording of Couple's Conversation. https://www.nytimes.com/2018/05/25/business/amazon-alexa-conversation-shared-echo.html/, 2018.

[33] Meet STEMosaur! https://cognitoys.com/, 2018.

[34] SmartThinQ with VoiceAssistants. https://www.lg.com/us/support/works-with-google-alexa-voice-assistant, 2018.

[35] Tesla's Siri integration now works with Model 3. https://electrek.co/2018/03/20/teslas-siri-integration-now-works-with-model-3/, 2018.

[36] Volkswagen now lets Apple users unlock their cars with Siri. https://www.theverge.com/2018/11/12/18087416/volkswagen-vw-car-net-app-siri-shortcuts, 2018.

[37] Was the Alexa Butt Dial a Big Deal? Steps You Can Take if You Are Concerned. https://voicebot.ai/2018/05/28/was-the-alexa-butt-dial-a-big-deal-steps-you-can-take-if-you-are-concerned/, 2018.

[38] Amazon's Alexa Allegedly Calls 911 During Domestic Violence Incident. https://wflanews.iheart.com/featured/pm-tampa-bay-with-ryan-gorman/content/2019-07-17-amazons-alexa-allegedly-calls-911-during-domestic-violence-incident/, 2019.

[39] DeepSpeech 0.6: Mozilla's Speech-to-Text Engine Gets Fast, Lean, and Ubiquitous . https://hacks.mozilla.org/2019/12/deepspeech-0-6-mozillas-speech-to-text-engine/, 2019.

[40] syzbot. https://syzkaller.appspot.com/upstream, 2019.

[41] DeepSpeech Slow Inference Speed Raspberry Pi 3B . https://github.com/mozilla/DeepSpeech/issues/3000/, 2020.

[42] Raspberry Pi 4 vs Raspberry Pi 3B+. https://magpi.raspberrypi.org/articles/raspberry-pi-4-vs-raspberry-pi-3b-plus//, 2020.

[43] Alexa for PC . https://developer.amazon.com/en-US/alexa/devices/pcs, 2021.

[44] Alexa skill market . https://www.amazon.com/alexa-skills/b?ie=UTF8&node=13727921011, 2021.

[45] Amazon Alexa. https://en.wikipedia.org/wiki/Amazon_Alexa, 2021.

[46] Amazon Echo. https://en.wikipedia.org/wiki/Amazon_Echo, 2021.

[47] Apple Platform Security - Secure Enclave. https://support.apple.com/guide/security/secure-enclave-sec59b0b31ff/web, 2021.

[48] Apple Platform Security - Touch ID and Face ID security. https://support.apple.com/guide/security/touch-id-and-face-id-security-sec067eb0c9e/web, 2021.

[49] AVS Device SDK. https://developer.amazon.com/alexa-voice-service/sdk, 2021.

[50] Cortana Devices SDK. https://developer.microsoft.com/en-us/cortana/devices, 2021.

[51] CVE Details. Linux Kernel: Vulnerability Statistics. https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html, 2021.

[52] CVE Details. Op-tee: Vulnerability Statistics. https://www.cvedetails.com/product/56969/Linaro-Op-tee.html, https://www.cvedetails.com/product/42749/Linaro-Op-tee.html, https://www.cvedetails.com/product/36161/Op-tee-Op-tee-Os.html, 2021.

[53] CVE Details. Windows 10: Vulnerability Statistics. https://www.cvedetails.com/product/32238/Microsoft-Windows-10.html, 2021.

[54] CVE Details. XEN: Vulnerability Statistics. https://www.cvedetails.com/product/23463/XEN-XEN.html, 2021.

[55] Every Alexa command you can give your Amazon Echo smart speaker. https://www.cnet.com/how-to/every-alexa-command-you-can-give-your-amazon-echo-smart-speaker/, 2021.

[56] Firejail Security Sandbox. https://firejail.wordpress.com/, 2021.

[57] Google Assistant. https://assistant.google.com, 2021.

[58] Google Assistant SDK for devices. https://developers.google.com/assistant/sdk/, 2021.

[59] HomePod. https://en.wikipedia.org/wiki/HomePod, 2021.

[60] Invoke. https://en.wikipedia.org/wiki/Invoke_(smart_speaker), 2021.

[61] Laugh factory joke bank. http://www.laughfactory.com/jokes/, 2021.

[62] Mi AI Speaker. https://www.mi.com/aispeaker/, 2021.

[63] Mozzila Common Voice. https://voice.mozilla.org/en//, 2021.

[64] Offensive/Profane Word List . http://www.cs.cmu.edu/~biglou/resources/bad-words.txt, 2021.

[65] Personal Digital Assistant – Cortana Home Assistant. https://www.microsoft.com/en-us/cortana, 2021.

[66] Pico Text-to-Speech. https://www.openhab.org/addons/voice/picotts/, 2021.

[67] Siri – Apple. https://www.apple.com/siri/, 2021.

[68] Slot Type Reference. https://developer.amazon.com/en-US/docs/alexa/custom-skills/slot-type-reference.html/, 2021.

[69] Total number of Amazon Alexa skills from January 2016 to September 2019 . https://www.statista.com/statistics/912856/amazon-alexa-skills-growth/, 2021.

[70] Understand How Users Invoke Custom Skills. https://developer.amazon.com/en-US/docs/alexa/custom-skills/understanding-how-users-invoke-custom-skills.html/, 2021.

[71] Voice Assistant on Fitbit Smartwatches . https://www.fitbit.com/global/us/technology/voice, 2021.

[72] What is WordNet? https://wordnet.princeton.edu/, 2021.

[73] H. Abdullah, W. Garcia, C. Peeters, P. Traynor, K. R. B. Butler, and J. Wilson. Practical Hidden Voice Attacks Against Speech and Speaker Recognition Systems. 2019.

[74] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation For UNIX Development. In *Proc. Summer 1986 USENIX Conference*, 1986.

[75] R. Aloufi, H. Haddadi, and D. Boyle. Privacy Preserving Speech Analysis Using Emotion Filtering at the Edge: Poster Abstract. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, SenSys '19, 2019.

[76] R. Aloufi, H. Haddadi, and D. Boyle. Privacy-preserving Voice Analysis via Disentangled Representations. In *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*, pages 1–14, 2020.

[77] Amazon Alexa. Create the Interaction Model for Your Skill. https://developer.amazon.com/docs/custom-skills/create-the-interaction-model-for-your-skill.html, 2018.

[78] A. Amiri Sani and T. Anderson. The Case for I/O-Device-as-a-Service. In *Proc. ACM HotOS*, 2019.

[79] A. Amiri Sani, K. Boos, M. Yun, and L. Zhong. Rio: A System Solution for Sharing I/O between Mobile Systems. In *Proc. ACM MobiSys*, 2014.

[80] N. Asmussen, M. Roitzsch, and H. Härtig. M^3x: Autonomous Accelerators via Context-Enabled Fast-Path Communication. In *Proc. ACM ASPLOS*, 2019.

[81] N. Asmussen, M. Völp, B. Nöthen, H. Härtig, and G. Fettweis. M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores. In *Proc. ACM ASPLOS*, 2016.

[82] A. Athalye, A. Belay, M. Kaashoek, R. Morris, and N. Zeldovich. Notary: A device for secure transaction approval. In *Proc. ACM SOSP*, 2019.

[83] P. Athe and S. Dasgupta. A Comparative Study of 6T, 8T and 9T Decanano SRAM cell. In *Proc. IEEE Symposium on Industrial Electronics & Applications*, 2009.

[84] A. M. Azab, K. Swidowski, J. M. Bhutkar, W. Shen, R. Wang, and P. Ning. SKEE: A Lightweight Secure Kernel-level Execution Environment for ARM. In *Proc. ACM MobiSys*, 2016.

[85] M. Backes, S. Bugiel, S. Gerling, and P. von Styp-Rekowsky. Android Security Framework: Enabling Generic and Extensible Access Control on Android. *arXiv preprint arXiv:1404.1395*, 2014.

[86] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A. Sadeghi, and E. Stapf. CURE: A Security Architecture with CUstomizable and Resilient Enclaves. In *Proc. USENIX Security Symposium*, 2021.

[87] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough Static Analysis of Device Drivers. In *Proc. ACM EuroSys*, 2006.

[88] A. Baumann, P. Barham, P. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proc. ACM SOSP*, 2009.

[89] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. *Proc. USENIX OSDI*, 2014.

[90] T. Blasing, L. Batyuk, A.-D. Schmidt, S. Camtepe, and S. Albayrak. An Android Application Sandbox System for Suspicious Software Detection. In *MALWARE*, 2010.

[91] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *Proc. USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[92] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *OSDI*, 2008.

[93] G. Campagna, R. Ramesh, S. Xu, M. Fischer, and M. S. Lam. Almond: The Architecture of an Open, Crowdsourced, Privacy-Preserving, Programmable Virtual Assistant. In *Proc. ACM WWW*, 2017.

[94] N. Carlini, P. Mishra, V. T., Y. Zhang, M. Sherr, C. Shields, D. Wagner, and W. Zhou. Hidden Voice Commands. In *Proc. of 25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[95] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto. SoK: Understanding the Prevailing Security Vulnerabilities in Trustzone-assisted TEE Systems. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2020.

[96] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on Binary Code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012.

[97] S. K. Cha, M. Woo, and D. Brumley. Program-Adaptive Mutational Fuzzing. In *IEEE Symposium on Security and Privacy*, 2015.

[98] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[99] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proc. ACM Asia-Pacific Workshop on Systems (APSys)*, 2011.

[100] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *Proc. USENIX Security*, 2014.

[101] T. Chen, L. Shangguan, Z. Li, and K. Jamieson. Metamorph: Injecting Inaudible Commands into Over-the-air Voice Controlled Systems. 2020.

[102] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports. Overshadow: a Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proc. ACM ASPLOS*, 2008.

[103] Y. Chen, Y. Li, L. Lu, Y. Lin, H. Vijayakumar, Z. Wang, and X. Ou. Instaguard: Instantly deployable hot-patches for vulnerable system programs on android. In *Proc. Internet Society NDSS*, 2018.

[104] Y. Chen, Y. Zhang, Z. Wang, L. Xia, C. Bao, and T. Wei. Adaptive android kernel live patching. In *Proc. USENIX Security Symposium*, 2017.

[105] L. Cheng, C. Wilson, S. Liao, J. Young, D. Dong, and H. Hu. Dangerous Skills Got Certified: Measuring the Trustworthiness of Skill Certification in Voice Personal Assistant Platforms. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, 2020.

[106] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: a Platform for In-Vivo Multi-Path Analysis of Software Systems. In *Proc. ACM ASPLOS*, 2011.

[107] V. Chipounov, V. Kuznetsov, and G. Candea. The S2E Platform: Design, Implementation, and Applications. *ACM Trans. Comput. Syst.*, 2012.

[108] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An Empirical Study of Operating Systems Errors. In *Proc. ACM SOSP*, 2001.

[109] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling Dynamic Program Analysis from Execution in Virtual Environments. In *USENIX Annual Technical Conference*, 2008.

[110] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *USENIX Security*, 2004.

[111] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. In *Proc. ACM CCS*, 2017.

[112] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end Containment of Internet Worms. In *Proc. ACM Symp. Operating Systems Principles*, 2005.

[113] A. Costin, J. Zaddach, A. Francillon, D. Balzarotti, and S. Antipolis. A Large-Scale Analysis of the Security of Embedded Firmwares. In *Proc. USENIX Security Symposium*, 2014.

[114] N. V. Database. CVE-2021-0200: Out-of-bounds write in the firmware for Intel(R) Ethernet 700 Series Controllers before version 8.2 may allow a privileged user to potentially enable an escalation of privilege via local access. `https://nvd.nist.gov/vuln/detail/CVE-2021-0200`.

[115] N. V. Database. Vulnerability summary for cve-2015-6639.

[116] D. Davidson, B. Moench, S. Jha, and T. Ristenpart. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *USENIX Security*, 2013.

[117] T. Denning, T. Kohno, and H. Levy. Computer Security and the Modern Home. *Communications of the ACM*, 56, 2013.

[118] W. Diao, X. Liu, Z. Zhou, and K. Zhang. Your Voice Assistant is Mine: How to Abuse Speakers to Steal Information and Control Your Phone. In *Proc. of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices (SPSM)*, 2014.

[119] D. Dubois, R. Kolcun, A. Mandalari, M. Paracha, D. Choffnes, and H. Haddadi. When Speakers Are All Ears: Characterizing Misactivations of IoT Smart Speakers. *Proceedings on Privacy Enhancing Technologies*, 2020.

[120] K. Elphinstone and G. Heiser. From L3 to seL4 What Have We Learnt in 20 Years of L4 Microkernels? In *Proc. ACM SOSP*, 2013.

[121] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI*, 2010.

[122] D. R. Engler, M. F. Kaashoek, and J. O. Jr. Exokernel: an Operating System Architecture for Application-Level Resource Management. In *Proc. ACM SOSP*, 1995.

[123] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *Proc. ACM EuroSys*, 2006.

[124] E. Fernandes, J. Jung, and A. Prakash. Security Analysis of Emerging Smart Home Applications. In *Proc. of 37th IEEE Symposium on Security and Privacy*, 2016.

[125] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash. FlowFence: Practice Data Protection for Emerging IoT Application Frameworks. In *Proc. of 25th USENIX Security Symposium*, 2016.

[126] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proc. ACM SOSP*, 2017.

[127] K. Franz. Add a microSD Slot with the Pmod MicroSD. `https://digilent.com/blog/add-a-microsd-slot-with-the-pmod-microsd/`, 2021.

[128] A. Frumusanu. Huawei Announces Mate 40 Series: Powered by 15.3bn Transistors 5nm Kirin 9000. `https://www.anandtech.com/show/16156/huawei-announces-mate-40-series`, 2020.

[129] W. Futral and J. Greene. *Intel Trusted Execution Technology for Server Platforms: A Guide to More Secure Datacenters*. Apress Media LLC, Springer Nature, 2013.

[130] Q. Ge, Y. Yarom, T. Chothia, and G. Heiser. Time Protection: The Missing OS Abstraction. In *Proc. ACM EuroSys*, 2019.

[131] A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. J. Elphinstone, V. Uhlig, J. E. Tidswell, L. Deller, and L. Reuther. The SawMill Multiserver Approach. In *Proc. ACM SIGOPS European workshop: beyond the PC: new challenges for the operating system*, 2000.

[132] Y. GmbH. SymbiYosys (sby) Documentation. `https://symbiyosys.readthedocs.io/en/latest/index.html`, 2021.

[133] P. Godefroid, M. Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *NDSS*, 2008.

[134] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. Cache Attacks on Intel SGX. In *Proc. ACM European Workshop on Systems Security (EuroSec)*, 2017.

[135] R. Grisenthwaite. Arm CCA will put confidential compute in the hands of every developer. `https://www.arm.com/company/news/2021/06/arm-cca-will-put-confidential-compute-in-the-hands-of-every-developer`, 2021.

[136] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom. Another Flip in the Wall of Rowhammer Defenses. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2018.

[137] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. Wu, S. Weng, H. Zhang, and Y. Guo. Deep Specifications and Certified Abstraction Layers. In *Proc. ACM POPL*, 2015.

[138] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proc. USENIX OSDI*, 2016.

[139] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. Liblit. Eio: Error handling is occasionally correct. In *Proc. FAST*, 2008.

[140] A. Hern. Apple contractors 'regularly hear confidential details' on Siri recordings. `https://www.theguardian.com/technology/2019/jul/26/apple-contractors-regularly-hear-confidential-details-on-siri-recordings`, 2019.

[141] F. Hetzelt and R. Buhren. Security Analysis of Encrypted Virtual Machines. In *Proc. ACM VEE*, 2017.

[142] S. Heuser, A. Nadkarni, W. Enck, and A. Sadeghi. ASM: A Programmable Interface for Extending Android Security. In *Proc. USENIX Security Symposium*, pages 1005–1019, 2014.

[143] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. InkTag: Secure Applications on an Untrusted Operating System. In *Proc. ACM ASPLOS*, 2013.

[144] Z. Huang, M. D'Angelo, D. Miyani, and D. Lie. Talos: Neutralizing Vulnerabilities with Security Workarounds for Rapid Response. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2016.

[145] G. C. Hunt and J. R. Larus. Singularity: Rethinking the Software Stack. *ACM SIGOPS Operating Systems Review*, 2007.

[146] IBM. Software TPM Introduction. `http://ibmswtpm.sourceforge.net/ibmswtpm2.html`, 2021.

[147] S. Jain, V. Tiwari, A. Balasubramanian, N. Balasubramanian, and S. Chakraborty. PrIA: A Private Intelligent Assistant. In *Proc. ACM Workshop on Mobile Computing Systems & Applications (HotMobile)*, 2017.

[148] S. Jana, Y. Kang, S. Roth, and B. Ray. Automatically Detecting Error Handling Bugs Using Error Specifications. In *Proc. USENIX Security Symposium*, 2016.

[149] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated Atomicity-violation Fixing. In *Proc. ACM PLDI*, 2011.

[150] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceno, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proc. ACM SOSP*, 1997.

[151] A. Kadav, M. J. Renzelmann, and M. M. Swift. Fine-Grained Fault Tolerance using Device Checkpoints. In *ACM Proc. ASPLOS*, 2013.

[152] M. G. Kang, P. Poosankam, and H. Yin. Renovo: A Hidden Code Extractor for Packed Executables. In *Proceedings of the 2007 ACM Workshop on Recurring Malcode (WORM)*, 2007.

[153] Y. Kang, B. Ray, and S. Jana. Apex: Automated Inference of Error Specifications for C APIs. In *Proc. IEEE/ACM ASE*, 2016.

[154] S. Keil and C. Kolbitsch. Stateful fuzzing of wireless device drivers in an emulated environment. *Black Hat Japan*, 2007.

[155] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *Proc. ACM ISCA*, 2014.

[156] D. Kirat and G. Vigna. MalGene: Automatic Extraction of Malware Analysis Evasion Signature. In *ACM CCS*, 2015.

[157] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating Fuzz Testing. In *Proc. ACM CCS*, 2018.

[158] D. Kleidermacher, J. Seed, B. Barbello, S. Somogyi, and P. . T. s. t. Android. Pixel 6: Setting a new standard for mobile security. `https://security.googleblog.com/2021/10/pixel-6-setting-new-standard-for-mobile.html`, 2021.

[159] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proc. ACM SOSP*, 2009.

[160] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre Attacks: Exploiting Speculative Execution. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2019.

[161] D. Kumar, R. Paccagnella, P. Murley, E. Hennenfent, J. Mason, A. Bates, and M. Bailey. Skill Squatting Attacks on Amazon Alexa. In *Proc. of 27th USENIX Security Symposium*, 2018.

[162] V. Kuznetsov, V. Chipounov, and G. Candea. Testing Closed-Source Binary Device Drivers with DDT. In *Proc. USENIX Annual Technical Conference*, 2010.

[163] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *Proc. ACM EuroSys*, 2020.

[164] A. Lenharth, V. Adve, and S. T. King. Recovery Domains: An Organizing Principle for Recoverable Operating Systems. In *Proc. ACM ASPLOS*, 2009.

[165] J. Lettner, D. Song, T. Park, P. Larsen, S. Volckaert, and M. Franz. PartiSan: Fast and Flexible Sanitization via Run-time Partitioning. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, 2018.

[166] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, and P. Levis. Multiprogramming a 64 kB Computer Safely and Efficiently. In *Proc. ACM SOSP*, 2017.

[167] M. Li, Y. Zhang, Z. Lin, and Y. Solihin. Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In *Proc. USENIX Security Symposium*, 2019.

[168] S. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui. A Secure and Formally Verified Linux KVM Hypervisor. 2021.

[169] S. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui. Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor. In *Proc. USENIX Security Symposium*, 2021.

[170] S. Liao, C. Wilson, L. Cheng, H. Hu, and H. Deng. Measuring the Effectiveness of Privacy Policies for Voice Assistant Applications. In *Annual Computer Security Applications Conference*, ACSAC '20, 2020.

[171] J. Liedtke. Improving IPC by Kernel Design. *ACM SIGOPS Operating Systems Review*, 1993.

[172] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *Proc. USENIX Security Symposium*, 2016.

[173] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading Kernel Memory from User Space. In *Proc. USENIX Security Symposium*, 2018.

[174] H. Liu, Y. Wang, L. Jiang, and S. Hu. PF-Miner: A New Paired Functions Mining Method for Android Kernel in Error Paths. In *IEEE COMPSAC*, 2014.

[175] F. Long, S. Sidiroglou-Douskos, and M. Rinard. Automatic Runtime Error Repair and Containment via Recovery Shepherding. In *Proc. ACM PLDI*, 2014.

[176] K. Loughlin, S. Saroiu, A. Wolman, and B. Kasikci. Stop! Hammer Time: Rethinking Our Approach to Rowhammer Mitigations. In *Proc. ACM HotOS*, 2021.

[177] K. Lu, A. Pakki, and Q. Wu. Automatically Identifying Security Checks for Detecting Kernel Semantic Bugs. In *Proc. European Symposium on Research in Computer Security*, 2019.

[178] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna. DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers. In *Proc. USENIX Security Symposium*, 2017.

[179] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Proc. ACM EuroSys*, 2008.

[180] M. Mendonça and N. Neves. Fuzzing Wi-Fi Drivers to Locate Security Vulnerabilities. In *In IEEE European Dependable Computing Conference (EDCC)*, 2008.

[181] A. Meola. The digital trends disrupting the banking industry in 2021. `https://www.businessinsider.com/banking-industry-trends`, 2021.

[182] R. Mitev, M. Miettinen, and A. Sadeghi. Alexa Lied to Me: Skill-based Man-in-the-Middle Attacks on Virtual Assistants. In *Proc. ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, 2019.

[183] A. Moghimi, G. Irazoqui, and T. Eisenbarth. Cachezoom: How SGX Amplifies the Power of Cache Attacks. In *Proc. Springer International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2017.

[184] D. Moghimi, J. Van Bulck, N. Heninger, F. Piessens, and B. Sunar. CopyCat: Controlled Instruction-Level Attacks on Enclaves. In *Proc. USENIX Security Symposium*, 2020.

[185] D. Mukhopadhyay, M. Shirvanian, and N. Saxena. All your voices are belong to us: Stealing voices to fool humans and machines. In *Proc. of European Symposium on Research in Computer Security (ESORICS)*, 2015.

[186] S. Mutti, Y. Fratantonio, A. Bianchi, L. Invernizzi, J. Corbetta, D. Kirat, C. Kruegel, and G. Vigna. BareDroid: Large-Scale Analysis of Android Apps on Real Devices. In *Proc. Annual Computer Security Applications Conference (ACSAC)*, 2015.

[187] S. Naffziger, N. Beck, T. Burd, K. Lepak, G. Loh, M. Subramony, and S. White. Pioneering Chiplet Technology and Design for the AMD EPYC and Ryzen Processor Families: Industrial Product. In *Proc. ACM/IEEE ISCA*, 2021.

[188] R. Nandakumar, S. Gollakota, and N. Watson. Contactless Sleep Apnea Detection on Smartphones. In *Proc. ACM MobiSys*, 2015.

[189] V. Narayanan, T. Huang, D. Detweiler, D. Appel, Z. Li, G. Zellweger, and A. Burtsev. RedLeaf: Isolation and Communication in a Safe Operating System. In *Proc. USENIX OSDI*, 2020.

[190] L. Nelson, H. Sigurbjarnarson, K. Zhang, D. Johnson, J. Bornholt, E. Torlak, and X. Wang. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proc. ACM SOSP*, 2017.

[191] J. Newsome. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *NDSS*, 2005.

[192] J. Newsome, D. Brumley, and D. Song. Vulnerability-Specific Execution Filtering for Exploit Prevention on Commodity Software. In *Proc. Network and Distributed Systems Security Symposium (NDSS)*, 2006.

[193] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proc. IEEE ICSE*, 2013.

[194] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In *Proc. ACM SOSP*, 2009.

[195] S. Oh, H. Yoo, D. R. Jeong, D. H. Bui, and I. Shin. Mobile Plus: Multi-device Mobile Platform for Cross-device Functionality Sharing. In *Proc. ACM MobiSys*, 2017.

[196] R. Paccagnella, L. Luo, and C. W. Fletcher. Lord of the Ring(s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical. In *Proc. USENIX Security Symposium*, 2021.

[197] S. Pailoor, A. Aday, and S. Jana. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *Proc. USENIX Security Symposium*, 2018.

[198] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: Ten Years Later. In *Proc. ACM ASPLOS*, 2011.

[199] J. Pan, G. Yan, and X. Fan. Digtool: A Virtualization-Based Framework for Detecting Kernel Vulnerabilities. In *Proc. USENIX Security Symposium*, 2017.

[200] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Proc. ACM SOSP*, 2009.

[201] A. Phaneuf. State of mobile banking in 2020: top apps, features, statistics and market trends. `https://www.businessinsider.com/mobile-banking-market-trends`, 2019.

[202] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the Library OS from the Top Down. In *Proc. ACM ASPLOS*, 2011.

[203] M. Posner. How many ASIC Gates does it take to fill an FPGA? `https://blogs.synopsys.com/breakingthethreelaws/2015/02/how-many-asic-gates-does-it-take-to-fill-an-fpga/`, 2015.

[204] Quarklab. BREAKING SAMSUNG'S ARM TRUSTZONE. `https://i.blackhat.com/USA-19/Thursday/us-19-Peterlin-Breaking-Samsungs-ARM-TrustZone.pdf`, 2019.

[205] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos. Flip Feng Shui: Hammering a Needle in the Software Stack. In *Proc. USENIX Security Symposium*, 2016.

[206] M. J. Renzelmann, A. Kadav, and M. M. Swift. SymDrive: Testing Drivers without Devices. In *Proc. USENIX OSDI*, 2012.

[207] N. Roy, H. Hassanieh, and R. Choudhury. BackDoor: Making Microphones Hear Inaudible Sounds. In *Proc. of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2017.

[208] N. Roy, S. Shen, H. Hassanieh, and R. Choudhury. Inaudible Voice Commands: The Long-Range Attack and Defense. In *Proc. of 15th Symposium on Networked Systems Design and Impalementation (NSDI)*, 2018.

[209] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau. Error propagation analysis for file systems. In *Proc. ACM PLDI*, 2009.

[210] S. Saha, J. Lozi, G. Thomas, J. L. Lawall, and G. Muller. Hector: Detecting Resource-Release Omission Faults in Error-Handling Code for Systems Software. In *Proc. IEEE/IFIP DSN*, 2013.

[211] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *Proc. USENIX Security Symposium*, 2017.

[212] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Proc. Springer International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2017.

[213] S. M. Seyed Talebi, A. Amiri Sani, S. Saroiu, and A. Wolman. MegaMind: A Platform for Security & Privacy Extensions for Voice Assistants. In *in Proc. ACM MobiSys*, 2021.

[214] S. M. Seyed Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. Amiri Sani, and Z. Qian. Charm: Facilitating Dynamic Analysis of Device Drivers of Mobile Systems. In *Proc. USENIX Security Symposium*, 2018.

[215] S. M. Seyed Talebi, Z. Yao, A. Amiri Sani, Z. Qian, and D. Austin. Undo Workarounds for Kernel Bugs. In *Proc. USENIX Security Symposium*, 2021.

[216] S. Shankland. Apple's A15 Bionic chip powers iPhone 13 with 15 billion transistors, new graphics and AI. `https://www.cnet.com/tech/mobile/apples-a15-bionic-chip-powers-iphone-13-with-15-billion-transistors-new-graphics-and-ai/`, 2021.

[217] F. Shezan, H. Hu, J. Wang, G. Wang, and Y. Tian. Read Between the Lines: An Empirical Measurement of Sensitive Applications of Voice Personal Assistant Systems. In *Proceedings of The Web Conference 2020*, WWW '20. Association for Computing Machinery, 2020.

[218] H. Shi, R. Wang, Y. Fu, M. Wang, X. Shi, X. Jiao, H. Song, Y. Jiang, and J. Sun. Industry Practice of Coverage-guided Enterprise Linux Kernel Fuzzing. In *Proc. ACM European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2019.

[219] Y. Shizuku, T. Hirose, N. Kuroki, M. Numa, and M. Okada. A 24-transistor static flip-flop consisting of nors and inverters for low-power digital vlsis. In *Proc. IEEE International New Circuits and Systems Conference (NEWCAS)*, 2014.

[220] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Proc. Network and Distributed Systems Security Symposium (NDSS)*, 2015.

[221] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. ASSURE: Automatic Software Self-healing Using REscue points. In *Proc. ACM ASPLOS*, 2009.

[222] H. Sigurbjarnarson, L. Nelson, B. Castro-Karney, J. Bornholt, E. Torlak, and X. Wang. Nickel: A framework for Design and Verification of Information Flow Control Systems. In *Proc. USENIX OSDI*, 2018.

[223] A. Simpson, F. Roesner, and T. Kohno. Securing Vulnerable Home IoT Devices with an In-Hub Security Manager. In *Proc. of 1st International Workshop on Pervasive Smart Living Spaces (PerLS)*, 2017.

[224] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz. SoK: Sanitizing for Security. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2019.

[225] A. Sotirov. Hotpatching and the rise of third-party patches. In *Black Hat Technical Security Conference*, 2006.

[226] V. Strumpen. Introduction to Digital Circuits: Basic Digital Circuits. http://bibl.ica.jku.at/dc/build/html/basiccircuits/basiccircuits.html, 2015.

[227] R. Tao, J. Yao, X. Li, S. Li, J. Nieh, and R. Gu. Formal Verification of a Multiprocessor Hypervisor on Arm Relaxed Memory Hardware. In *Proc. ACM SOSP*, 2021.

[228] T. C. G. (TCG). TCG PC Client Specific TPM Interface Specification (TIS), Specification Version 1.3. https://trustedcomputinggroup.org/wp-content/uploads/TCG_PCClientTPMInterfaceSpecification_TIS__1-3_27_03212013.pdf, 2013.

[229] T. C. G. (TCG). TPM 2.0 Library. https://trustedcomputinggroup.org/resource/tpm-library-specification/, 2019.

[230] D. Team. NEWS: OmniPod Tubeless Insulin Pump to Offer Smartphone Control Soon. https://www.healthline.com/diabetesmine/omnipod-smartphone-control-diabetes, 2019.

[231] Y. Tian and B. Ray. Automatically Diagnosing and Repairing Error Handling Bugs in C. In *Proc. ACM ESEC/FSE*, 2017.

[232] J. Tucek, J. Newsome, S. Lu, C. Huang, S. Xanthos, D. Brumley, Y. Zhou, and D. Song. Sweeper: A Lightweight End-to-end System for Defending Against Fast Worms. 2007.

[233] B. Ur, J. Jung, and S. Schechter. The Current State of Access Control for Smart Devices in Homes. In *Proc. of Workshop on Home Usable Privacy and Security (HUPS)*, 2013.

[234] T. Vaidya, Y. Zhang, M. Sherr, and C. Shields. Cocaine Noodles: Exploiting the Gap Between Human and Machine Speech Recognition. In *In Proc. of the 9th USENIX Conference on Offensive Technologies (WOOT)*, 2015.

[235] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. FORESHADOW: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proc. USENIX Security Symposium*, 2018.

[236] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *Proc. ACM CCS*, 2016.

[237] J. Vander Stoep. Android: Protecting the Kernel. In *Linux Security Summit (LSS)*, 2016.

[238] A. Vasudevan, S. Chaki, P. Maniatis, L. Jia, and A. Datta. üBERSPARK: Enforcing Verifiable Object Abstractions for Automated Compositional Security Analysis of a Hypervisor. In *Proc. USENIX Security Symposium*, 2016.

[239] R. Wang, A. M. Azab, W. Enck, N. Li, P. Ning, X. Chen, W. Shen, and Y. Cheng. SPOKE: Scalable Knowledge Collection and Attack Surface Analysis of Access Control Policy for Security Enhanced Android. In *Proc. ACM ASIA CCS*, 2017.

[240] T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *IEEE Symposium on Security and Privacy*, 2010.

[241] D. Weber, A. Ibrahim, H. Nemati, M. Schwarz, and C. Rossow. Osiris: Automated Discovery of Microarchitectural Side Channels. In *Proc. USENIX Security Symposium*, 2021.

[242] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *Proc. ACM ISSTA*, 2010.

[243] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically Finding Patches Using Genetic Programming. In *Proc. IEEE ICSE*, 2009.

[244] S. Weiser and M. Werner. SGXIO: Generic Trusted I/O Path for Intel SGX. In *Proc. ACM CODASPY*, 2017.

[245] L. Wilke, J. Wichelmann, M. Morbitzer, and T. Eisenbarth. SEVurity: No Security Without Integrity: Breaking Integrity-Free Memory Encryption with Minimal Assumptions. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2020.

[246] C. Willems, T. Holz, and F. Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security & Privacy*, 2007.

[247] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley. Scheduling Black-box Mutational Fuzzing. In *ACM CCS*, 2013.

[248] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In *Proc. USENIX Security Symposium*, 2016.

[249] Xilinx. Xilinx Standalone Library Documentation. OS and Libraries Document Collection. UG643 (v2021.1) June 16, 2021.

[250] Xilinx. Zynq UltraScale + Device. Technical Reference Manual. UG1085 (v2.2) December 4, 2020.

[251] Z. Xu, Y. Zhang, L. Zheng, L. Xia, C. Bao, Z. Wang, and Y. Liu. Automatic Hot Patch Generation for Android Kernels. In *Proc. USENIX Security Symposium*, 2020.

[252] B. Yadegari and S. Debray. Symbolic Execution of Obfuscated Code. In *ACM CCS*, 2015.

[253] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A Generic Approach to Automatic Deobfuscation of Executable Code. In *IEEE Symposium on Security and Privacy*, 2015.

[254] L. K. Yan and H. Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *USENIX Security*, 2012.

[255] Q. Yan, K. Liu, Q. Zhou, H. Guo, and N. Zhang. SurfingAttack: Interactive Hidden Attack on Voice Assistants Using Ultrasonic Guided Waves. 2020.

[256] Y. Yan, Z. Li, Q. A. Chen, C. Wilson, T. Xu, E. Zhai, Y. Li, and Y. Liu. Understanding and Detecting Overlay-based Android Malware at Market Scales. In *Proc. ACM MobiSys*, 2019.

[257] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *ACM CCS*, 2007.

[258] X. Yuan, Y. Chen, Y. Zhao, Y. Long, X. Liu, K. Chen, S. Zhang, H. Huang, X. Wang, and C. Gunter. CommanderSong: A Systematic Approach for Practical Adversarial Voice Recognition. In *Proc. of 27th USENIX Security Symposium*, 2018.

[259] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti. Avatar: A framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *Proc. Network and Distributed Systems Security Symposium (NDSS)*, 2014.

[260] E. Zeng, S. Mare, and F. Roesner. End User Security & Privacy Concerns with Smart Homes. In *Proc. of the 13th USENIX Conference on Usable Privacy and Security (SOUPS)*, 2017.

[261] E. Zeng and F. Roesner. Understanding and Improving Security and Privacy in Multi-User Smart Homes: A Design Exploration and In-Home User Study. In *Proc. USENIX Security Symposium*, 2019.

[262] G. Zhang, C. Yan, X. Ji, T. Zhang, T. Zhang, and W. Xu. DolphinAttack: Inaudible Voice Commands. In *CCS 2017*, 2017.

[263] H. Zhang, D. She, and Z. Qian. Android Root and its Providers: A Double-Edged Sword. In *Proc. ACM CCS*, 2015.

[264] N. Zhang, X. Mi, X. Feng, X. Wang, Y. Tian, and F. Qian. Dangerous Skills: Understanding and Mitigating Security Risks of Voice-Controlled Third-Party Functions on Virtual Personal Assistant Systems. In *Proc. of the IEEE Symposium on Security and Privacy*, 2019.

[265] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou. TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices. *IACR Cryptology ePrint Archive*, 2016:980, 2016.

[266] X. Zhang, Y. Zhang, J. Li, Y. Hu, H. Li, and D. Gu. Embroidery: Patching Vulnerable Binary Code of Fragmentized Android Devices. In *IEEE ICSME*, 2017.

[267] Y. Zhang, X. Luo, and H. Yin. DexHunter: Toward Extracting Hidden Code from Packed Android Applications. In *ESORICS*, 2015.

[268] Y. Zhang, L. Xu, A. Mendoza, G. Yang, P. Chinprutthiwong, and G. Gu. Life after Speech Recognition: Fuzzing Semantic Misinterpretation for Voice Assistant Applications. In *Proc. Internet Society NDSS*, 2019.

Appendix A

Bugs description

Here we discuss a few of realworld vulnerabilites and unpatched real bugs that we use in evaluating bowknots in chapter 3. In this appenix we manually inspect the effectivness of Hecaton in automatically generating bowknots for these bugs.

CVE-2019-2293 This vulnerability, which is rated as medium security severity, is caused by a possible null pointer dereference in Qualcomm camera `ife` module. A null pointer dereference might happen because of lack of a proper check on the `isp_resource` length variable before calling `cam_ife_mgr_acquire_hw_for_ctx()`. There are 7 functions in this bug's call stack. OctopOS overall generates 10 undo statements in these functions. OctopOS successfully detects several types of state-mutating statements and their corresponding undo statements including direct function calls, function pointers, and global variable assignments. However, our manual investigation shows that one bowknot does not correctly undo the side effect of its function. In `cam_context_handle_acquire_dev()` function `ioctl_ops.acquire_dev()`, which modifies the state of the camera device driver is called, but it is not paired with its undo function, `ioctl_ops.release_dev()` . OctopOS missed

this statement because the original error handling code was not complete and did not call `ioctl_ops.release_dev()`.

After correcting the incomplete bowknot manually, when we run the PoC of this vulnerability on the mitigated kernel, all bowknots in the call stack get executed and successfully undo the side effects of the PoC. The camera device remains functional after this successful undo.

CVE-2019-1999 In function `binder_alloc_free_page()`, there is a possible double-free vulnerability due to improper locking. This vulnerability is rated as high security severity because it could lead to local escalation of privilege in the kernel with no additional execution privileges needed. In 2 functions in the call stack, there are 2 state-mutating statements, which OctopOS automatically detects and uses to generate bowknots. Our manual investigation shows that the generated bowknots are complete. Also, OctopOS-generated bowknots preserve the binder's functionality after recovery. Hence, after the recovery, the system is functional and successfully passes a binder test program that we execute. Our test program consists of two processes, a binder-server and a binder-client. It checks for successful communication between these two processes.

CVE-2019-10529 This is a use-after-free bug that can get triggered with a race condition while attempting to mark the entry pages as dirty using the function `set_page_dirty()`. Use-after-free bugs in the kernel can cause a system crash, put the system in an unexpected state, or be used in privilege escalation exploits. Automated bowknots generated by OctopOS mitigate this vulnerability and preserve the GPU driver's functionality. To test the GPU driver's functionality, we used the "GPU Mark BenchMark" application, which tests the GPU under the stress of rendering. We do not notice any difference in the result before and after OctopOS mitigates this vulnerability. Our manual investigation also shows that bowknots undo worked correctly in this case.

CVE-2019-2000 This is a bug in the binder module of the Pixel3 phone. There are

4 functions in this bug's call stack. OctopOS finds 6 state-mutating statements in these functions and generate the undo code for them in their bowknots. Our experiments show that the binder module remains functional after triggering this bug and executing the bowknots. Our manual investigation confirms that there are no other statements that result in any change in the system's state, which leaks to non-local variables.

CVE-2019-2284: This is a bug in camera driver of Pixel3 phone. There are 4 functions in this bug's call stack. OctopOS finds 10 state-mutating statements in these functions and generates the undo code for them. However, our experiments show that the Camera device loses its functionality after triggering this bug and executing the bowknots. Our investigation shows that 2 out of 4 bowknots OctopOS generates for this bug's functions are incomplete. In `cam_sensor_core_power_up()` function, there is a loop in which it turns on an array of voltage regulators. Although this function has another `for` loop in its error handling path which turns off the same array of the voltage regulators, OctopOS currently does not support multi-statement undo, and only produces a warning for the user. Our investigation shows that the bowknot generated for `cam_sensor_driver_cmd()` is also not complete. In this case, OctopOS fails to generate the complete bowknot because of the incomplete error handling code. Please note that after we manually add the missing undo statements to the mentioned functions, the system and the camera device remain functional after triggering the bug and execution of bowknots.

Syzbot bug a11372b6c9b5fd4abe1c266903bcb27e80e8f2bc

This is a bug in the TTY driver of the x86 Linux-Next kernel. There are 5 functions in this bug's call stack. OctopOS locates two state-mutating functions and generates proper undo code for them. It pairs `kmalloc()` with `kfree()` and `console_lock()` with `console_unlock()` in the `con_font_get()` function. The system and TTY module remain functional after triggering this bug and execution of bowknots. Our manual investigation shows that in one of the functions, `fbcon_get_font()`, there are changes to a data structure called `font`, which

is not a local variable of `fbcon_get_font()` and is provided as an input variable. Since there is no undo code to revert changes of the `font` data structure, at first glance, it seems that the bowknot does not completely undo the driver's state. However, our further analysis shows that `font` data structure is not a global variable of the driver and is defined as a local variable in `con_font_get()` function, which is the parent function of `fbcon_get_font()`. As a result, changes to the `font` data structure do not leak to the other parts of the kernel before bowknot's execution. Hence, our manual investigation shows that OctopOS-generated bowknots correctly undo the effects of partially-executed system call, which confirms the result of the functionality test.

Syzbot bug 9ad0eb3691bac24fd21ae3d8add8c08014a69f57

This is a bug in the file system of the upstream x86 Linux kernel. There are 10 functions in this bug's call stack. OctopOS finds one state-mutating statement and pairs it with its undo statement. This pair is `blk_start_plug()` and `blk_finish_plug()`, which appears twice in the execution path of this function. The file system functionality tests, including kernel self-tests for the file system, successfully pass after triggering the bug and execution of bowknots. In two functions in the call stack, we observe statements that change the non-local variables of those functions. However, similarly to the previous case, our detailed analysis shows that these non-local variables are not part of the global state of the system or the file system; they are local variables defined in the parent functions in the call stack. There is no change to the system's state, which does not have undo code in the bowknots. As a result, our manual investigation is in agreement with the functionality test.

Syzbot bug d708485af9edc3af35f3b4d554e827c6c8bf6b0f

This is a bug in HCI Bluetooth driver of the x86 Linux-Next kernel. There are three functions in the call stack of this bug. OctopOS successfully pairs 4 state-mutating statements with their undo code in these functions' bowknots. We test the functionality of HCI Bluetooth driver with a user-space program that uses this driver and with the network stack self-tests of

Linux kernel. The HCI Bluetooth driver and the network stack remain functional after triggering the bug and execution of bowknots. Our manual investigation shows that, in addition to the 4 state-mutating statements that OctopOS finds, there are three other function calls that can potentially change the state of the system. One is `hci_req_cmd_complete()`, which manipulates the `hdev` the driver data structure. However, our further analysis shows that this function does not get executed in the execution path of this bug. As a result, it is not a concern. The two other function calls, which can possibly change the state of system, are `hci_send_to_sock()` and `hci_send_to_monitor()`. Sending data over HCI socket changes the state of system and it is not reversible. However, our deeper analysis shows in the case of triggering this bug, these two functions return at the beginning and do not reach to the point that they change the state of system. As a result, the success of functionality test indicates the correct undo of system state in this case, too.

Syzbot bug f0ec9a394925aafbdf13d0a7e6af4cff860f0ed6

This is a bug in a network driver of the upstream x86 Linux kernel. The bug is located in HCI Bluetooth driver. There are 11 functions in this bug's call stack. Although OctopOS generates complete bowknot for 10 out of the 11 functions in the call stack for this bug, the remaining incomplete bowknot results in unsuccessful recovery. The last function in the call stack of this bug, the `__list_add()` function, is designed to add an entry to a specified location of a doubly linked list in the kernel. It modifies the two nodes that it wants to insert a new node in between. The bug occurs after processing of the first node but before the second node. At this point, the doubly link list is corrupted and there is no code to undo this corruption. We could not fix this problem in the two-hour window that we allow for manual work for each bug.

Syzbot bug 0d93140da5a82305a66a136af99b088b75177b99

This is a bug in a network driver of the upstream x86 Linux kernel. The bug is located in HCI physical layer driver. There are 11 functions in this bug's call stack. OctopOS pairs

5 state-mutating statements with their undo code in these function's bowknots. However, the network self-test result changes after triggering the bug and execution of the bowknots. Hence, the functionality test for the automatically-generated bowknot fails for this function. Our investigation shows that there is one pair of state-mutating and undo functions, which OctopOS missed because of its database's incompleteness. When we manually add `hci_conn_drop()` to the bowknot of the function `hci_phy_link_complete_evt()` to reverse the effect of `hci_conn_hold()`, the bowknots become complete and the functionality test passes successfully.