UC Irvine
UC Irvine Electronic Theses and Dissertations

Title
Securing Operating Systems with Dynamic Program Analysis

Permalink
https://escholarship.org/uc/item/2xw264d4

Author
Hung, Hsin-Wei

Publication Date
2023

Peer reviewed|Thesis/dissertation

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

UNIVERSITY OF CALIFORNIA,
IRVINE

Securing Operating Systems with Dynamic Program Analysis

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Engineering

by

Hsin-Wei Hung

Dissertation Committee:
Associate Professor Ardalan Amiri Sani, Chair
Assistant Professor Joshua Garcia
Assistant Professor Zhou Li

2023

Chapter 2 © 2020 ACM
Chapter 3 © 2022 ACM
All other materials © 2023 Hsin-Wei Hung

DEDICATION

To my wife, Diane, and my family for your love and support

TABLE OF CONTENTS

Page

LIST OF FIGURES **vi**

LIST OF TABLES **viii**

ACKNOWLEDGMENTS **ix**

VITA **x**

ABSTRACT OF THE DISSERTATION **xii**

1 Introduction **1**
- 1.1 Security of Operating Systems . . . 1
- 1.2 Dynamic Program Analysis . . . 2
- 1.3 Symbolic Execution . . . 3
- 1.4 Hardening Operating Systems . . . 5
- 1.5 Fuzzing . . . 6

2 Mousse: A System for Selective Symbolic Execution of Programs with Untamed Environments **8**
- 2.1 Introduction . . . 8
- 2.2 Background & Motivation . . . 11
  - 2.2.1 Selective Symbolic Execution . . . 11
  - 2.2.2 Program Environment . . . 14
- 2.3 Challenges & Design . . . 15
- 2.4 Environment-Aware Path Concurrency . . . 17
- 2.5 Path Offloading & Distributed Execution . . . 22
  - 2.5.1 Path Offloading . . . 23
  - 2.5.2 Environment-Forced Symbolic Variables . . . 26
- 2.6 Analysis . . . 26
  - 2.6.1 Android I/O Services . . . 27
  - 2.6.2 Target Analyses . . . 27
- 2.7 Implementation . . . 29
- 2.8 Evaluation . . . 31
  - 2.8.1 Performance . . . 31
  - 2.8.2 Coverage . . . 34

2.8.3 Analysis Results 35
2.9 Other Related Work 37

3 Sifter: Protecting Security-Critical Kernel Modules in Android through Attack Surface Reduction 40
3.1 Introduction 40
3.2 Overview 44
3.2.1 Motivating Examples 44
3.2.2 Goals 45
3.2.3 Feasibility 46
3.2.4 Remaining Research Questions 48
3.3 Threat Model 49
3.4 Workflow 49
3.5 Filter Policies 52
3.5.1 Argument Limiting Policy 52
3.5.2 Operation Serialization Policy 53
3.5.3 Operation Deprecation Policy 56
3.6 Implementation & Prototype 57
3.6.1 Checking File Descriptors 58
3.6.2 Deploying Complex Filters using eBPF 58
3.6.3 Preventing Collusion Attacks 59
3.6.4 Preventing TOCTOU Attacks 60
3.6.5 Training Programs 60
3.7 Effectiveness Study 61
3.7.1 Mimicry Attacks 66
3.8 Evaluation 67
3.8.1 False Positive Analysis 67
3.8.2 Performance 69
3.8.3 Energy Consumption 72
3.8.4 Training Time 72
3.9 Discussions 73
3.10 Related Work 74

4 BRF: eBPF Runtime Fuzzer 77
4.1 Introduction 77
4.2 Background 80
4.2.1 Workflow of eBPF 80
4.2.2 eBPF Verifier 82
4.2.3 BPF Runtime 82
4.3 Overview 84
4.3.1 Goals 84
4.3.2 Design 85
4.3.3 Workflow 85
4.4 Generating Semantic-Correct eBPF Programs 87
4.4.1 Helper Function Availability 89

4.4.2 Helper Function Arguments 90
4.4.3 Variable Safety Checks 91
4.4.4 Program Context Access 93
4.4.5 Reference 94
4.5 eBPF program dependencies 94
4.5.1 Creating Compatible Maps 95
4.5.2 Relocating eBPF programs 97
4.6 Executing eBPF programs 97
4.6.1 Program attachment 98
4.6.2 Triggering the hooks 98
4.7 Implementation 99
4.7.1 Collecting Coverage of eBPF Programs 99
4.8 Evaluation 101
4.8.1 Fuzzing Effectiveness 101
4.8.2 Discovered Vulnerabilities 105
4.9 Discussions on Future Maintenance of BRF 108
4.10 Related Work 109

5 Conclusion **113**

Bibliography **115**

LIST OF FIGURES

Page

2.1 *Simple hypothetical program used to describe the inner workings of SSE.* 12
2.2 *Mousse design.* 16
2.3 *Different SSE designs.* 16
2.4 *A real code example demonstrating the importance of environment-aware concurrency. We have modified and eliminated parts of the code for clarity.* 18
2.5 *Environment consistency for concurrent path execution. (Left) All paths to be executed in a device. (Middle) Environmentally consistent and inconsistent paths after a state-mutating ecall. (Right) Paths after the second state-mutating ecall.* 21
2.6 *Simple hypothetical program used to demonstrate the offloading strategy in Mousse.* 24
2.7 *Impact of environment-aware concurrency on execution time.* 32
2.8 *Impact of distributed execution on execution time.* 33
2.9 *Code coverage for different APIs of Android I/O services. Conc. and M-II refer to concrete execution and Mousse with concretization II, respectively.* . . 36

3.1 *Simplified code illustrating CVE-2017-9682.* 44
3.2 *Sifter's workflow. The blue shapes are part of Sifter and the white ones are external.* 50
3.3 *The Sifter's filter limits syscalls of untrusted programs.* 51
3.4 *The numbers of triggerable vulnerabilities in $V_{syscall}$ before and after applying filters.* 63
3.5 *Simplified code illustrating CVE-2016-2468.* 64
3.6 *Simplified code illustrating CVE-2018-13905.* 65
3.7 *False positives of filters trained and tested using different numbers of training programs.* 67
3.8 *Performance overhead of Sifter on Binder micro-benchmarks.* 70
3.9 *Performance overhead of Sifter on Binder macro-benchmark (SurfaceFlinger-Stress).* 71
3.10 *Performance overhead of Sifter on GPU macro-benchmarks (PM1: Passmark 3D simple, PM2: Passmark 3D complex, PM3: Passmark OpenGL ES).* 72

4.1 The design of BRF 85
4.2 The workflow of BRF, where the orange blocks denote the mutated inputs . 86

4.3 *Simplified code showing how the verifier checks if a helper is available to kprobe eBPF programs* . . . 89
4.4 *Simplified code showing how the verifier checks if a helper is available to kprobe eBPF programs* . . . 90
4.5 *An example of a semantic-correct eBPF program rejected by the verifier.* . . . 93
4.6 *The success rate of BRF and Syzkaller in covering helper functions with a given number of arguments.* . . . 104
4.7 *Coverage of eBPF runtime components of BRF and Syzkaller under 40-hour fuzzing sessions* . . . 105
4.8 *Simplified code illustrating a Proof-of-Concept (PoC) of CVE-2022-2905.* . . . 106
4.9 *lockdep warning showing how CVE-2023-0160 could cause a deadlock.* . . . 107

LIST OF TABLES

Page

2.1 *Single-API testing of operating system services with Mousse. Abbreviations used in the table: GS = GPU Stack, AP = AudioProvider, CS = CameraServer, Res. = Resource constraint, Env. = Environment consistency.* 34

3.1 *Information of filters generated by Sifter.* P_{Arg}*is the number of argument limiting policies,* $P_{Op.S}$*is the number of operations in the operation serialization policy, and* $P_{Op.D}$*is the number of syscalls allowed by the operation deprecation policy. #Inst. is the number of eBPF instructions.* 58

3.2 *Training programs used for generating syscall policies.* 61

3.3 *Analysis of CVEs in Binder and Qualcomm KGSL GPU driver (CVE-20YY-XXXX is shortened to* $YY\text{-}XXXX^{T}$*, where T is the vulnerability type – 1: information leakage, 2: OOB access, 3: data race, 4: kernel API misuse, and 5: logical bug).* 62

4.1 *Types of error messages in the eBPF verifier.* 87

4.2 *Comparison of programs loaded, attached, and triggered by Syzkaller and BRF under 40-hour fuzzing sessions. *It is derived by dividing the total number of programs executed by total programs attached.* 101

4.3 *Expressiveness of eBPF program generated and successfully loaded by Syzkaller and BRF.* 101

4.4 *Summary of vulnerabilities discovered by BRF (OOB access: out-of-bound access).* 105

ACKNOWLEDGMENTS

I am extremely thankful to my advisor, Professor Ardalan Amiri Sani, for mentoring my research and personal growth.

I would also like to thank my labmates, Drs. (and future Drs.) Yingtong Liu, Zhihao (Zephyr) Yao, Yuxin (Myles) Liu, Mingyi Chen, and Joseph Bursey for their help, support, and stimulating discussions. I specifically like to thank my collaborator on multiple projects, Dr. Yingtong Liu for her immense efforts. I would also like to thank Professor Joshua Garcia and Professor Zhou Li for serving on my dissertation committee. Thank you for your insight and feedback.

Finally, the endeavor would not have been possible without my wife and my family. It is the structure and love you bring that got me through.

VITA

Hsin-Wei Hung

EDUCATION

Doctor of Philosophy in Computer Engineering **2023**
University of California, Irvine *Irvine, California*

Master of Science in Electrical Engineering **2017**
National Tsing Hua University *Hsinchu, Taiwan*

Bachelor of Science in Electrical Engineering **2014**
National Tsing Hua University *Hsinchu, Taiwan*

RESEARCH EXPERIENCE

Graduate Research Assistant **2017–2023**
University of California, Irvine *Irvine, California*

REFEREED CONFERENCE PUBLICATIONS

Mousse: A System for Selective Symbolic Execution of Programs with Untamed Environments — **Apr 2020**
ACM EuroSys

Sifter: Protecting Security-Critical Kernel Modules in Android through Attack Surface Reduction — **Oct 2022**
ACM MobiCom

SOFTWARE

Mousse — `https://trusslab.github.io/mousse/`
EuroSys'20

Sifter — `https://trusslab.github.io/sifter/`
MobiCom'22

BRF — `https://trusslab.github.io/brf/`

ABSTRACT OF THE DISSERTATION

Securing Operating Systems with Dynamic Program Analysis

By

Hsin-Wei Hung

Doctor of Philosophy in Computer Engineering

University of California, Irvine, 2023

Associate Professor Ardalan Amiri Sani, Chair

Operating systems are critical components of the Trusted Computing Base (TCB) of any computer system. They need to be secured to ensure the confidentiality, integrity and availability of the applications running on top of them. To achieve this, dynamic program analysis techniques have been used to find vulnerabilities and mitigate weaknesses in operating systems. By executing the program under test and performing analysis, dynamic program analysis can achieve higher accuracy compared to static analysis. This dissertation showcases three applications of dynamic program analysis and addresses the challenges of using them to secure operating systems.

First, we present Mousse, a solution to address the challenges of using Selective Symbolic Execution (SSE) for analyzing the Hardware Abstraction Layer (HAL) programs. HAL programs, which communicate between user space programs and the kernel, are an integral part of many operating systems. Existing symbolic execution engines are inefficient in testing them due to false positives introduced by symbolic hardware or poor performance caused by expensive memory synchronization. In Mousse, we proposed process-level SSE. Combining with environment-aware concurrency and distributed execution, it can achieve good performance with no false positives.

Secondly, we present Sifter, a syscall filtering framework for tightening the attack surface

of the operating system. Since malware often uses abnormal syscall patterns to exploit vulnerabilities in the kernel, we build a framework that can trace and analyze syscalls from legitimate programs, and then generate syscall filters that only allow syscall patterns seen in the traces.

Finally, we present the eBPF runtime fuzzer (BRF) for fuzzing eBPF runtime components in the Linux kernel. eBPF is a modern technology in the Linux kernel that allows programs supplied from the user space to be executed in the kernel. Although fuzzing has been proved to be effective in finding bugs in the Linux kernel, the state-of-the-art fuzzer, Syzkaller, fails to reach the eBPF runtime components due to 1) syscall dependencies in the eBPF subsystem, 2) semantic rules imposed by the eBPF verifier to ensure the safety of eBPF programs, and 3) failure to attach and execute the loaded programs. With extensive experiments, we show that BRF is not only more efficient in reaching runtime components, but also covers more code. Finally, it manages to find 4 vulnerabilities (3 of which have been assigned CVE numbers) in the eBPF runtime components.

Chapter 1

Introduction

1.1 Security of Operating Systems

Operating systems are critical components in the Trusted Computing Base (TCB) of any computer system. Computer systems normally consist of three basic components, computer hardware, software programs that provide services to end users, and operating system sitting between the hardware and software. The operating system orchestrates the hardware and software by handling the input/output (IO), managing the resources (e.g., memory), and scheduling the execution of tasks. Since the operating system possesses great control and visibility into the system, they are frequently targeted in cyber-attacks.

While the design of operating systems provides basic protection for the kernel, there are still ways to break the protection. To prevent normal programs from arbitrarily accessing the system resources managed by the kernel, operating systems implement dual mode operation to separate the execution of processes into user mode and kernel mode. While programs operating in user mode (i.e., user space processes) can execute normally and access their own memory, they cannot perform privileged operations such as modifying page tables, handling

interrupts, or accessing hardware registers. If normal programs want to perform privileged operations, they will need to request such operations to be performed by the kernel through system calls. However, due to the complexity of operating systems, bugs inevitably make their way to the kernel. As a result, attackers can trick the kernel into entering invalid states or accessing invalid resources by calling series of maliciously crafted syscalls that trigger bugs. If the consequences pose security implications, these bugs become vulnerabilities. By exploiting vulnerabilities and compromising operating systems, the attacker can perform attacks such as denial-of-service (DOS), information leakage, or even modifying arbitrary data if they gain the highest privilege. Therefore, researchers and engineers have been working on advanced techniques to secure operating systems.

1.2 Dynamic Program Analysis

Dynamic program analysis is a set of techniques that analyze programs as they execute. By combining instrumentation, tracing, profiling, and emulation techniques, it can control the execution and observe the behavior of programs.

Dynamic program analysis, compared to static analysis, is more accurate. Without executing a program in a realistic environment, static analysis often loses its precision in the process of estimating the interaction of the environment and reasoning if a execution path is reachable. As a result, static analysis often produces more false positives.

Dynamic program analysis is a powerful tool that can be used to help secure the operating system, and there are two major directions. First, it can be used to secure the operating system by finding bugs or vulnerabilities so that they can be fixed. Two common techniques that do so are symbolic execution [88, 89, 45, 52, 53, 54, 30, 75, 19, 20, 118, 38, 35, 36, 37, 119, 78] and fuzzing [117, 61, 107, 32, 18, 33, 83, 86, 99]. The second direction is to

help harden the operating system by making exploitation of vulnerabilities harder. In this dissertation, we explore techniques in both directions and address the challenges when using them to secure the operating system.

1.3 Symbolic Execution

Symbolic execution is a dynamic program analysis technique that has the ability to reason about the input required to reach a given execution path. It works by first marking inputs or internal variables of interest as symbolic variables. Then, as the program executes, the symbolic variables are propagated. If the execution meets a branch instruction where the predicate contains symbolic variables, the execution will be forked into two, and a corresponding constraint will be added to each execution path. Finally, when the execution reaches the point of interest, by solving the path constraints, the set of inputs required to reach the current state can be derived.

While symbolic execution is powerful, it suffers from the issue of path explosion. Since the execution will fork whenever there is a conditional branch to explore both paths, when analyzing large-scale programs, it could result in forking many execution paths. To alleviate the issue, selective symbolic execution (SSE) has been proposed [35]. By only executing programs in symbolic mode when inside the code region of interest, SSE reduces the numbers of paths to be explored during the execution.

Besides, existing symbolic execution engines are not efficient when one wants to analyze programs with untamed environments, that is, programs coupled with hardware that cannot be easily modeled nor virtualized. This is very common as mobile devices and computers nowadays are equipped with special hardware (e.g., camera, sensors, accelerators) in addition to general-purpose hardware (e.g., CPU, memory, and storage). Hardware Abstraction Layer

(HAL) programs are examples of programs with untamed environments. They facilitate the development of user space programs building on top of the hardware by communicating with the drivers in the kernel on behalf of the user space programs.

To analyze programs with untamed environments, there are two existing approaches. First, *symbolic environment* works by running the program without the actual environment. Whenever the program interacts with the untamed environment, a symbolic value will be created and returned, which significantly exacerbates the path explosion issue. The second approach, *decoupled SSE* also runs the program on an SSE engine, but it also runs another concrete execution engine with the actual environment. When the program wants to interact with the environment, memory states will need to be synchronized between the two engines and then the concrete engine will transfer the result back to the symbolic execution engine. When the program frequently interacts with the environment, memory synchronization will introduce significant overhead to the performance.

To tackle this problem, we presented Mousse [73], a system for selective symbolic execution of programs with untamed environments. The key idea of Mousse is to have an integrated symbolic and concrete execution engine running as an OS process on the actual environment. This allows us to interact with the real environment without expensive memory synchronization and avoiding creating more symbolic variables. Besides, to improve the performance, we further proposed environment-aware concurrency so that multiple paths can be executed simultaneously without corrupting the consistency of the environment. Furthermore, we developed distributed execution to further increase the performance by distributing the path exploration to multiple devices. We discuss Mousse's design and how it helps with finding bugs in Chapter 2.1.

1.4 Hardening Operating Systems

Dynamic program analysis can also help build runtime enforcement rules by analyzing valid program behaviors. For example, control flow integrity (CFI) works by analyzing possible control flow transfers offline. Then, during runtime, invalid control flow transfers will not be permitted. As a result, certain attacks that rely on hijacking the control flow such as return oriented programming (ROP) can be mitigated. Another example, SPOKE [111], generates SEAndroid (an access control framework on Android) policies by analyzing the existing functional test suites as they carry rich semantics.

In Chapter 3, we present Sifter [62], a framework for protecting security-critical kernel modules in Android through attack surface reduction. Based on the observation that malware often use weird syscall patterns to exploit vulnerabilities in the kernel, we aim to filter the syscalls of untrusted programs by only allowing the patterns that we learn from legitimate programs. However, to make filters effective, they need to have the ability to inspect deeply into arguments stored in the user space. Besides, they also need to support stateful filter policies so that syscalls can be checked based on previous syscalls since a syscall with legitimate arguments but within an invalid sequence can be used to exploit a vulnerability. Moreover, we aim to minimize the amount of manual effort needed by the analyst, i.e., the effort to study the syscall interface and develop the filter for every kernel modules.

Sifter addresses the aforementioned problems with the help of two recent advancements, eBPF and Syzkaller [11]. eBPF [100, 101] is a quickly evolving technology in the Linux kernel that provides general programmability to the Linux kernel during runtime. By allowing programs supplied by the user to be loaded and attached to different locations in the kernel, it greatly increases the flexibility and performance of deploying customized logic in the kernel. It also provides persistent storage, i.e., eBPF maps, that last across invocations. We use eBPF for tracing legitimate programs and enforcing filtering rules, so that the filter not only

can support deep argument inspection but also stateful policies. To address the problem of manual effort, we leverage the knowledge of the state-of-the-art syscall fuzzer, Syzkaller. To efficiently fuzz the syscall interface, Syzkaller provides a syscall description language and has the description of many kernel modules built in it. Reusing this knowledge, Sifter can automatically generate the structure of filters and greatly reduces the manual effort. We discuss the design and evaluation in more details in Chapter 3.

1.5 Fuzzing

Fuzzing is a dynamic program analysis technique mainly used to find bugs and vulnerabilities. It works by feeding randomly generated inputs to the program under test and checking for unexpected output. For example, Syzkaller tries to catch error messages produced by kernel sanitizers such as, KASAN [6], KUBSAN [3], and KCSAN [2], when they discover bugs. Differential fuzzing catches errors by comparing the outputs of two functionally identical programs.

Fuzzing relies on feedbacks from the program execution to guide the exploration of the execution paths. Known as black box testing, fuzzing often has limited information about the execution, which is unlike symbolic execution that is able to derive the inputs required for specific execution paths. Therefore, to deeply explore different execution paths, it relies on feedback signals from the execution to guide the exploration. More specifically, by selecting an input that reaches code region of interest, mutating the input, and execute the mutated input, a fuzzer will be able to explore the program better. Among different feedback strategies, coverage gives the most direct feedback, i.e., how much code an input covers.

Although fuzzing has shown to be successful by finding many vulnerabilities in the past, we found that it is inefficient in testing the eBPF runtime components. Working on Sifter, we

are also interested in knowing whether the eBPF subsystem is safe since it is in the TCB of Sifter. Therefore, we used Syzkaller to fuzz the eBPF subsystem, which in the past has found many bugs in the eBPF verifier, the checker in the kernel to safeguard kernel from executing malicious eBPF programs. However, we discovered that it is not efficient in testing the runtime components that will be used once eBPF programs pass the verifier. There are three reasons behind the inefficiency. First, multiple syscalls are required to load an eBPF program to the kernel and they are dependant to the eBPF program. However, Syzkaller only generates these syscalls randomly, which causes eBPF programs to be rejected. Second, the eBPF verifier imposes strict semantics on the programs in addition to the bytecode syntax. While Syzkaller is aware of the syntax of eBPF bytecode, it does not have knowledge about the semantics. Finally, eBPF programs, once loaded (i.e., passing the verifier), need to be attached to the entry points in the kernel and be triggered in order to be executed. Since syscalls are randomly generated in Syzkaller, it is not guaranteed to attach and trigger the programs.

To address these problems, we present the eBPF runtime fuzzer (BRF). By studying the eBPF verifier, we developed dependency-aware and semantic-aware input generation/mutation logic for BRF so that eBPF programs can be successfully loaded. In addition, BRF triggers the execution of eBPF programs using two different strategies. In Chapter 4, we discuss BRF in more detail.

Chapter 2

Mousse: A System for Selective Symbolic Execution of Programs with Untamed Environments

2.1 Introduction

Selective symbolic execution (SSE) is a powerful program analysis technique that can analyze multiple execution paths of a program. As in symbolic execution, when the analyst marks a variable as symbolic (i.e., capable of taking any arbitrary concrete value), the SSE engine executes and analyzes all program paths possible for different values of the variable. In order to avoid the path explosion that comes with symbolic execution, the analyst can configure the SSE engine to execute parts of the program in concrete mode, i.e., normal execution with concrete variables.

In the past, SSE has been used to implement various types of analysis, such as bug and vulnerability detection [70, 36, 90], performance profiling [36] and reverse-engineering of

binaries [34, 36]. In addition, it can be used for taint analysis, hybrid fuzzing [102, 118], and for exploit generation and analysis [19, 20].

In this dissertation, we address a critical challenge that hinders the applicability of SSE to a large and important set of programs: *programs with untamed environments.* In order to analyze multiple paths within a program, SSE runs multiple *forks*, or instances, of the program, one per path, in order to execute conditional statements with symbolic predicates. To eliminate interference between the execution of these program instances, each uses a separate instance of the program's environment. Two common approaches are modeling the program's environment in software [96, 19, 20, 30] and virtualizing it [36]. Unfortunately, neither approach is feasible for *untamed* environments, i.e., those that include diverse hardware components and their device drivers. Examples include operating system services managing I/O devices (i.e., I/O services), libraries (such as GPU-specific OpenGL/ES, OpenCL, and CUDA libraries), and applications (such as vendor camera and telephony applications in smartphones). Modeling is infeasible, due to the complexity of the hardware components and their drivers; and virtualization is infeasible too, because such hardware components do not support it.

The research community has explored two approaches. The first uses a symbolic environment [70, 34, 90], i.e., all the return values from the environment are marked as symbolic (since the correct environment of the program is not available). This approach results in path explosion and false code coverage, as it executes program paths that would not execute when actual return values from the real program environment are used. The second approach, *decoupled SSE*, is to allow the symbolic execution engine to communicate with a concrete execution engine running on the actual environment of the program [119, 78, 9]. This approach has noticeable overhead, due to the overhead of memory state transfers between the two engines.

In Mousse, we tackle this challenge with three solutions. First, we present a novel SSE

design, called *process-level SSE* (here, a process refers to an operating system process), which integrates the symbolic and concrete execution engines in the same operating system process containing the program. This allows both engines to easily interact with the underlying environment. Moreover, both engines use a unified memory, which eliminates the need to transfer the memory state between them, resulting in better performance. To support concurrent execution of program paths, process-level SSE executes each program path in a separate operating system process. Whenever the SSE engine explores a new path, it forks the current process and executes the new path in the child process. Forking a process is fast and efficient due to copy-on-write support in the kernel.

Second, we introduce *environment-aware concurrency* to allow multiple program paths to execute concurrently on top of the same environment, without observing inconsistent environment state. To do this, Mousse keeps track of the interactions of the different execution paths with the environment, and restricts the execution of environmentally inconsistent paths.

Third, while Mousse enables concurrent execution of multiple program paths in one device, the untamed environment fundamentally limits concurrency. This, and the fact that SSE is compute-heavy, means that analyzing complex programs, such as operating system services, takes a long amount of time. For example, testing a single API of an audio service with symbolic input in Pixel 3 takes our SSE engine 9 hours when using a single device. To address this problem, we introduce a distributed execution approach that supports concurrent execution of the analysis on multiple identical devices, while avoiding duplicate paths.

To demonstrate the benefits of Mousse, we use it to analyze five operating system services: two camera services, two audio services, and one graphics stack, in three smartphones, Pixel 3, Nexus 5X, and Nexus 5. We perform bug and vulnerability detection, searching for incorrect memory access and incorrect use of memory management APIs. We found two new crash bugs, and two new double-free vulnerabilities in these services. We also perform

taint analysis, to study the propagation of the inputs to the outputs of service APIs. We find that none of the APIs of this service, except for one, propagates its inputs to its outputs. This finding can be used to enhance the accuracy of taint analysis for programs that use these APIs. Moreover, we perform performance profiling of the Pixel 3 audio service, and find that it experiences 19% more L1 data cache misses for some playback configurations.

We perform extensive evaluation of Mousse. We show that Mousse's process-level SSE design reduces the execution time by at least 63% with respect to the state-of-the-art decoupled SSE. We also show that using a symbolic environment results in path explosion, which in turn prevents successful initialization of operating system services even after running the analysis for a few days. Our evaluation shows that Mousse's environment-aware concurrency and distributed execution help reduce the SSE execution time by up to 84% compared to running a single path at a time in one device.

We designed and built Mousse to analyze programs with untamed environments. However, we note that Mousse is capable of analyzing arbitrary programs, with high performance and ease. We have open sourced Mousse, so that others can leverage it in their analysis efforts [10].

2.2 Background & Motivation

2.2.1 Selective Symbolic Execution

Selective symbolic execution (SSE) is a powerful program analysis technique that can analyze multiple execution paths of a program [35, 36, 37, 119, 78, 9, 30, 75]. A path here refers to one in the control-flow graph of the program. Different inputs to the program may result in the execution of different paths, due to conditional statements. In SSE, similar

```
1  int prog_main(int arg_s, int arg_c) {
2    if (arg_s >= 13)
3      return func1(arg_s, arg_c);
4    else
5      return func2(arg_s, arg_c);
6  }
```

Figure 2.1: *Simple hypothetical program used to describe the inner workings of SSE.*

to symbolic execution, the analyst can mark a variable, including an input argument, as symbolic (i.e., with unknown concrete value); then the SSE engine executes the program paths corresponding to all possible values of the variable. In contrast to plain symbolic execution, the analyst can configure the SSE engine to execute some parts of the program in concrete mode, i.e., normal execution with concrete variables, in order to avoid path explosion.

We next use a simple example (Figure 2.1) to explain how SSE works. Assume that the analyst wishes to explore all the program paths that depend on the value of `arg_s`, but not those that depend on `arg_c`. She marks `arg_s` as symbolic, and assigns a concrete value to `arg_c`.

The SEE engine executes the program until it faces a conditional predicate with a symbolic variable (line 2). At this point, the execution *forks*, resulting in two instances of the program, each executing one of two resulting paths. The mechanism to fork the program depends on the SSE design, e.g., operating system process fork. Both paths now continue to use the symbolic variable, but they add constraints, derived from the conditional predicate. More specifically, one path executes the then-branch of the conditional (i.e., line 3) with the constraint `arg_s >= 13`. The other executes the else-branch (i.e., line 5) with the constraint `arg_s < 13`.

SSE supports *selective* symbolic execution. That is, parts of the program can be executed in concrete mode, which can help alleviate path explosion. Execution in concrete mode is similar to how a program normally executes. That is, the code in concrete mode does not

use symbolic variables; it can only use variables with concrete values. Therefore, no new paths are forked.

Assume the analyst has decided to execute `func1()` and `func2()` in concrete mode in the example. Once an execution path reaches either of these functions, the SSE engine switches from symbolic to concrete mode. Here, it needs to *concretize* any symbolic variables that are accessed by the code in concrete mode. In our example, `arg_s` needs to be concretized, as it is passed to these functions. To concretize a variable, the engine uses a solver to choose some concrete value that satisfies the path constraints. For instance, the solver might choose `arg_s = 14` when executing `func1()` in concrete mode.

Thus, the SSE engine is composed of two execution engines, *the symbolic execution engine* and *the concrete execution engine*. Both engines typically execute the program by *emulating the instructions in the program binary.* These engines need to communicate, e.g., to share the memory state when switching execution mode. Different SSE designs achieve this communication differently.

An SSE engine can support *concolic variables* as well. A concolic variable is a symbolic variable that also has a concrete value attached to it, called *concolic value*. The concolic value is used to determine which side of a conditional the path should take, when facing a symbolic predicate, in case forking is not needed. In the example, let us assume that the analyst marks `arg_s` as concolic with a concolic value of `20`. Moreover, the analyst configures the SSE engine to not fork at line 2. When it reaches this line, it branches by applying the concolic value to the predicate. If it evaluates to `true`, it executes the then-branch, otherwise the else-branch. In this example, since `20 >= 13` evaluates to true, the then-branch is executed.

2.2.2 Program Environment

The environment of a program is the set of all hardware and software components that it interacts with. This includes the operating system kernel (including device drivers) and hardware components. In SSE, the environment of the program is either modeled or virtualized, so that each forked program instance (executing a program path) can interact with a separate instance of the environment. For instance in the code sample in Figure 2.1, assume that `func1()` and `func2()` are syscalls. This means that both program paths interact with the underlying kernel. To make sure that these paths do not interfere with each other, their impact on kernel state must be isolated. One approach is to model the syscall [96, 19, 20, 30], i.e., to implement an approximation of its behavior in software, and to use that instead of the real syscall. Another approach is to virtualize the kernel and use a separate Virtual Machine (VM) for each path, forking the VM when forking the program path [36].

Unfortunately, there exists an important set of programs, whose environments cannot be easily modeled or virtualized. These are programs that interact with different hardware components and their drivers, such as I/O devices and accelerators. Modern mobile devices, such as smartphones, tablets, voice assistants, and VR/AR headsets, employ a large number of I/O devices, to stand out in a highly competitive market. For example, a smartphone might employ a powerful camera array [8] or an in-display fingerprint scanner [26]; a voice assistant may employ arrays of speakers and microphones for audio beamforming [82]; and a VR/AR headset may employ high-resolution displays, requiring powerful GPUs [115]. Data center servers, on the other hand, use various accelerators such as GPUs, TPUs, and FPGAs. This trend is fueled by the slowing down of Moore's law and is predicted to grow [29]. The programs that interact with these devices include operating system services (such as various I/O services in Android), libraries (such as GPU-specific OpenGL/ES, OpenCL, and CUDA libraries), and applications (such as customized vendor camera and telephony applications in smartphones).

Modeling the hardware and/or its device driver is a non-trivial task. Virtualizing the hardware is also non-trivial. Most hardware components, including I/O devices in mobile devices, do not support virtualization. The device assignment approach, which is often used to give a VM direct access to an I/O device [15, 71, 56, 23, 72], is not enough, as it does create multiple virtual instances of the device.

2.3 Challenges & Design

Our goal is to apply SSE to complex programs that interact with untamed environments. In this section, we introduce three challenges that we have faced in doing so, and our solutions to them.

Challenge I: direct access to the environment is critical. Since the untamed environment of a program cannot be modeled nor virtualized, the real environment must be used. To solve this, we introduce *process-level SSE*, an SSE design that enables both the symbolic and concrete execution engines to interact with the environment and to share the memory state. Thanks to this design, our SSE engine is the first to comprehensively analyze I/O services in Android. For more details of process-level SSE, please refer to the full paper [74].

Challenge II: path concurrency is feasible but requires environment-awareness. SSE execution is slow due to instruction emulation. To achieve acceptable performance, it is important to execute the program paths concurrently. However, the program's interaction with the environment creates a concurrency issue. This is because the environment is stateful (e.g., the state in a device driver or the underlying hardware component). If one program path mutates the environment state, other paths might receive unexpected responses from the environment. Therefore, we introduce *environment-aware concurrency*, a principled ap-

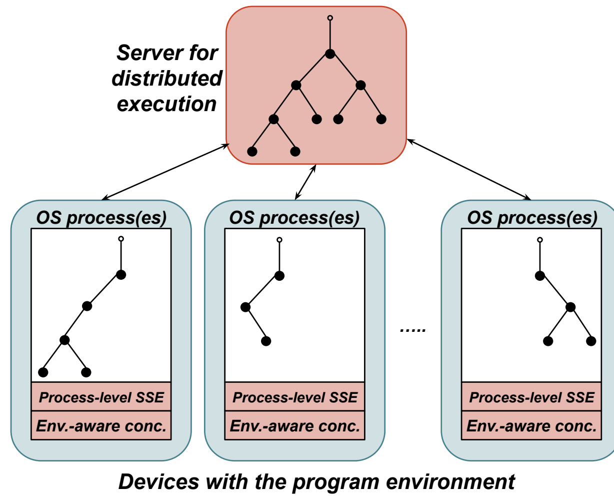


Figure 2.2: *Mousse design.*

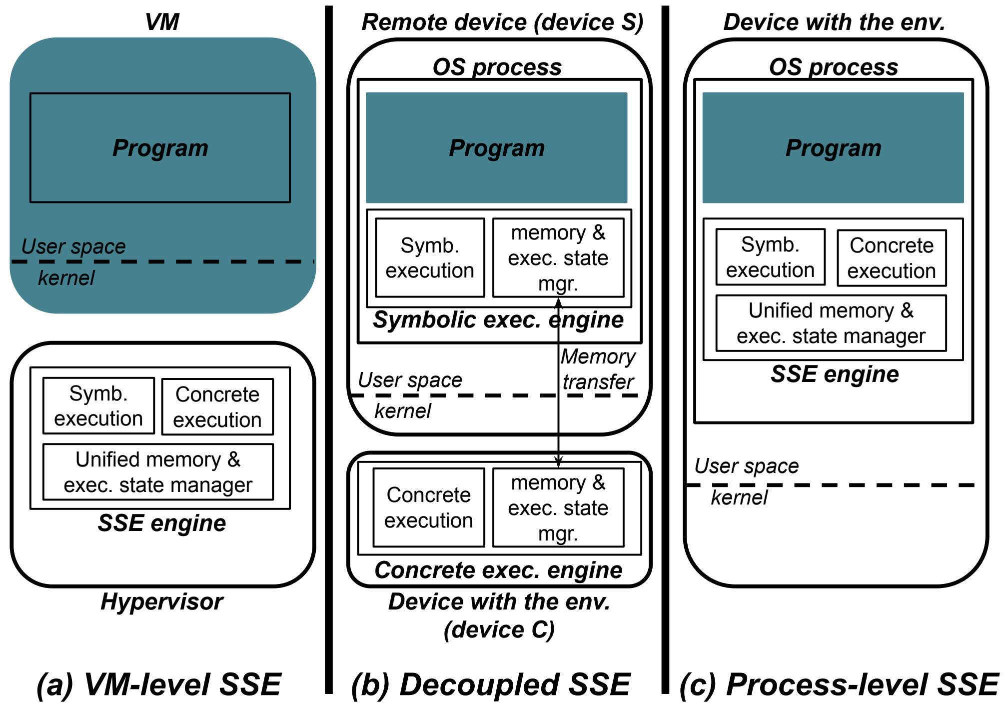


Figure 2.3: *Different SSE designs.*

proach to executing program paths concurrently while preventing inconsistent environment state from corrupting their execution.

Environment-aware concurrency, in the worst case, can result in sequential execution of all program paths. However, we show that an opportunity for concurrency exists when analyzing I/O services in Android: in the common case, multiple paths can execute concurrently. This is due to the fact that interactions with the environment are not frequent.

Challenge III: path concurrency might be limited but distributed execution helps. While Mousse enables concurrent execution of program paths, the degree of concurrency may be limited by the environment state. We show that distributed execution can address this performance bottleneck. To do so, when a device cannot execute a path, due either to the environment state or to resource constraint, it *offloads* the path to another device. Offloading a path refers to requesting a centralized server to assign the path to another device for execution.

Figure 2.2 illustrates the design of Mousse. It shows a centralized server distributing program paths to several devices, each of which uses process-level SSE and environment-aware concurrency to execute the paths. The server does not perform any analysis on the program itself. It acts as a simple work queue of paths waiting to be analyzed.

We next discuss the components of Mousse.

2.4 Environment-Aware Path Concurrency

SSE is slow as both symbolic and concrete execution engines emulate the instructions. Therefore, it is important to execute different program paths concurrently to speed up the execution. For example, in S^2E, whenever a path is forked, the whole VM is forked and the

```
/* Audio service out_write API */
1 static ssize_t out_write(struct audio_stream_out *stream, const void *buffer, size_t bytes) {
2   struct stream_out *out = (struct stream_out *)stream;
    ...
3   lock_output_stream(out); //This function calls pthread_mutex_lock(&out->lock);
    ...

4   long ns = (frames * (int64_t) NANOS_PER_SECOND) / out->config.rate;
5   request_out_focus(out, ns);
    ...
6   ret = pcm_write(out->pcm, (void *)buffer, bytes_to_write);
    ...
7   pthread_mutex_unlock(&out->lock);
    ...
8 }
```

```
/* Code in the audio driver where the error happens */
1 void *q6asm_is_cpu_buf_avail(int dir, struct audio_client *ac, uint32_t *size, uint32_t *index)
2 {
3   void *data;
4   unsigned char idx;
5   struct audio_port_data *port;
    ...
6   // dir 0: used = 0 means buf in use
7   // dir 1: used = 1 means buf in use
8   if (port->buf[idx].used == dir) {
9       // To make it more robust, we could loop and get the
10      // next avail buf, its risky though
11      pr_err("%s: Next buf idx[0x%x] not available, dir[%d]\n",
12       __func__, idx, dir);
13      mutex_unlock(&port->lock);
14      return NULL;
15  }
    ...
16 }
```

Figure 2.4: *A real code example demonstrating the importance of environment-aware concurrency. We have modified and eliminated parts of the code for clarity.*

resulting VM can run concurrently.

Key challenge. Unfortunately, for programs with untamed environments, blind concurrent execution can result in unexpected program behavior that would not happen in normal execution of the program. Given that the environment for a program cannot be modeled nor virtualized, the ecalls must be passed to the actual environment. Therefore, the concurrently executing program paths can impact each other's execution by mutating the state of the environment in unexpected ways. This state mutation is not problematic (and indeed desired) when only a single program path is executed, as in native execution of the program. However, when multiple paths are executed concurrently, some paths may see an inconsistent

environment state since all paths interact with the same environment.

Figure 2.4 illustrates why blind concurrency does not work using the example of a Pixel 3 audio service API. The API, called `out_write`, writes audio frames through the audio driver to the audio device. We perform SSE on this API by marking its inputs as symbolic. The execution forks multiple paths in function `request_out_focus` at line 5. Several of these paths then continue to call the `pcm_write` function at line 6, which issues an `ioctl` syscall to the audio driver to pass the audio frames. We observe that multiple paths receive an "out of memory" error from the driver, an unexpected behavior for these paths. On further investigation, the driver does not expect multiple concurrent writes. Indeed, this error would not normally happen due to the critical section in the `out_write` function in the audio service, which guarantees that the writes to the driver are sequential. Yet, the forking due to symbolic execution in the middle of this critical section results in an unexpected behavior.

In the figure, we also show the audio driver code (in the kernel) that returns the error. It checks if the DMA audio buffer is available for writing the data. In line 8, if the DMA buffer (`port->buf[idx]`) is being used, the function returns NULL (i.e., an error).

To address this challenge, Mousse keeps track of the interactions of different program paths with the environment. It prevents program paths from seeing inconsistent environment state.

We next define some terms and elaborate on our solution. A *state-mutating* ecall is one that, when executed, mutates the state of the environment in a way that *could* affect the execution of another path. Note that not all ecalls are state-mutating. For example, the execution of a memory allocation syscall in one path does not affect other paths since memory is virtualized. A *state-revealing* ecall is one that reveals the mutated state. Such an ecall returns a different result if a state-mutating syscall has been previously issued by another program path. In the previous example, the syscall to the audio driver is both state-mutating and state-revealing.

We assume that the analyst specifies which ecalls are state-mutating or state-revealing. In §2.7, we explain which ecalls we specify as such in our prototype.

Mousse splits the set of paths into *environmentally consistent* and *environmentally inconsistent* paths. Environmentally consistent paths are those whose execution is consistent with the state of the environment. In the beginning of the analysis and before the execution of any state-mutating ecalls, all paths are environmentally consistent. Environmentally consistent paths can execute with no restriction. Environmentally inconsistent paths are those whose execution is not consistent with the state of the environment, as a result of a state-mutating ecall issued by another path. Environmentally inconsistent paths *can also execute but their execution is restricted.*

The restriction enforced on environmentally inconsistent paths are two-fold. First, Mousse needs to prevent a path from seeing unexpected responses from the environment. Therefore, Mousse does not allow an environmentally inconsistent path to issue a state-revealing ecall, which may return unexpected responses due to the state-mutating ecall issued by some other path. Second, Mousse tries to prevent all paths from turning inconsistent. This is a heuristic designed to ensure some paths can fully finish their execution in the device. Therefore, Mousse does not allow an environmentally inconsistent path to issue a state-mutating ecall. If allowed, the state of the environment would be inconsistent with all executing paths.

Whenever a path issues a state-mutating ecall, it turns all other environmentally consistent paths into inconsistent ones. However, the paths that are later forked from this current path (i.e., children paths) remain environmentally consistent since they share the state-mutating ecall. Figure 2.5 illustrates this issue. Figure 2.5 (Left) shows the set of all paths, which are all environmentally consistent in the beginning. Figure 2.5 (Middle) shows what happens when one of the paths executes a state-mutating ecall. That path and its children remain consistent because the state-mutating ecall is part of their correct execution. However, the rest of the paths are turned inconsistent. Figure 2.5 (Right) shows what happens after a

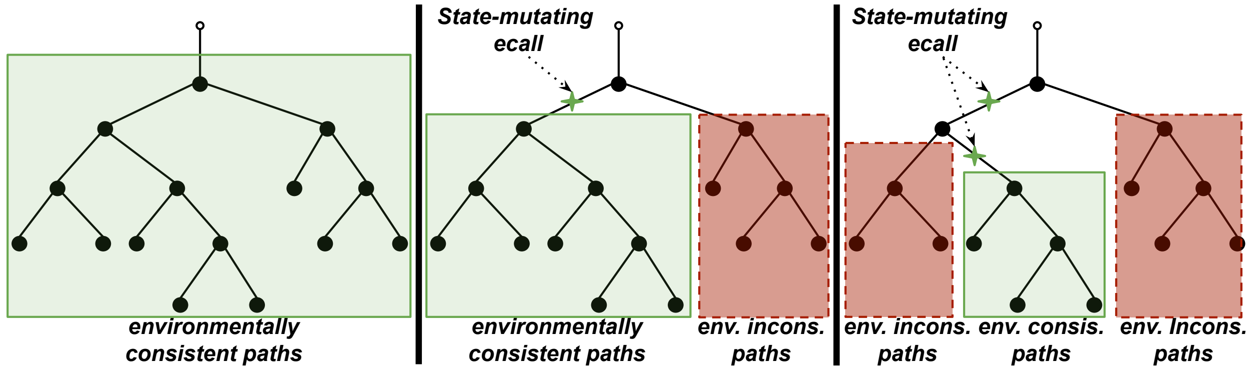


Figure 2.5: *Environment consistency for concurrent path execution. (Left) All paths to be executed in a device. (Middle) Environmentally consistent and inconsistent paths after a state-mutating ecall. (Right) Paths after the second state-mutating ecall.*

second state-mutating ecall. Similarly, the path executing this ecall and its children remain environmentally consistent, but the rest of the paths are turned inconsistent.

As mentioned, environmentally inconsistent paths can resume execution as long as they do not issue a state-mutating or state-revealing ecall. But if they attempt to execute one, Mousse suspends their execution and offloads them. §2.5 provides more details on the offloading process.

Mousse continues executing the paths until there are no other paths left that can be executed. At this point, it reboots the system to refresh the state of the environment. After the reboot, it contacts the server and ask for new paths to execute (§2.5).

Opportunity? Does concurrency provide any benefits in the presence of state-mutating and state-revealing ecalls? In other words, doesn't environment-aware concurrency simply result in sequential execution of all program paths? In §2.8.1, we show that even in the presence of such ecalls, concurrent execution can provide performance benefits. Moreover, when such calls are not present, Mousse's solution automatically increases the degree of concurrency.

2.5 Path Offloading & Distributed Execution

Mousse cannot execute all paths concurrently due to the environment state limitation (§2.4). Moreover, SSE is a time-consuming analysis due to instruction emulation and multi-path execution. For example, analyzing a single API call of an audio service with symbolic input in Pixel 3 takes 9 hours in our prototype when using a single smartphone with environment-aware path concurrency. To address this issue, Mousse adopts a distributed execution framework. That is, it distributes the program paths to multiple devices in order to reduce the execution time. In this section, we discuss our distributed execution strategy and our solution to an environment-related challenge.

Mousse's distributed execution strategy is dynamic and on-demand. That is, instead of assigning different program paths to different devices statically, it assigns one device to start the analysis. Then, if for some reason, some paths cannot be executed in that device, the paths are offloaded to a centralized server. The server does not perform any analysis on the program itself. It acts as a simple work queue for the devices to analyze different program paths. That is, devices, when idle, contact the server to download the program paths for execution.

Paths are offloaded from a device for two reasons: (*i*) inconsistent environment state, where the execution of one path makes the execution of another path infeasible (§2.4), and (*ii*) resource constraint, which limits the number of program paths that can be executed concurrently in a device. Currently, we set a fixed upper limit (determined empirically) for the total number of concurrent paths in one device. Alternatively, Mousse can dynamically monitor the resource consumption in the device to determine how many paths it can execute concurrently.

2.5.1 Path Offloading

The key component of distributed execution in Mousse is path offloading. Mousse performs path offloading using concolic program inputs. In SSE, one analyzes a program by marking its select inputs (e.g., API inputs or configuration options) as symbolic. During execution, whenever a path needs to be offloaded, Mousse solves the constraints on the path and generates a set of concrete values for program's symbolic inputs. It then offloads these values to the server. When the path is later downloaded by a device for execution, these concrete values can be used to mark the API inputs as concolic variables (§2.2.1), i.e., concolic inputs. The role of these concolic inputs is to guide the symbolic execution to re-execute the offloaded path from scratch.

One might wonder why Mousse does not offload the state of the execution of the path so that it does not need to be re-executed from scratch. The reason behind this is that the untamed environment state cannot be captured. This is because a hardware component and its driver might not provide an interface for taking snapshots of their state. Mousse's approach allows the path to re-execute from the beginning, which correctly reconstructs the environment state.

When a device downloads a path to execute, it performs the execution in two steps. In the first step, it uses the concolic inputs to execute the path from the beginning all the way to the point where the offload happened (i.e., the re-executed part of the path). In this part of the execution, no new paths will be forked. Instead, the concolic inputs are used to guide the execution through the conditional statements with symbolic predicates. In the second step, execution continues in the parts of the program that were not executed before (i.e., the new part of the path). When executing this part, forking is enabled and the concolic inputs are not needed anymore.

Disabling the forks in the re-executed part of the path is needed to avoid forking duplicate

```
1  int prog_main(int arg) {
2    if (arg >= 13) {
3      return syscall(SYSCALL_NR_1, ...); /* state-mutating */
4    } else {
5      int ret = syscall(SYSCALL_NR_2, ...); /* state-revealing */
6      if (arg <= 4)
7        return ret;
8      else
9        return func(ret);
10   }
11 }
```

Figure 2.6: *Simple hypothetical program used to demonstrate the offloading strategy in Mousse.*

paths. This re-execution itself is not problematic and it is in fact needed to recreate the state of the environment. However, if this re-executed part contains a conditional statement with a symbolic predicate and hence forks a new path, the fork would be a duplicate and hence the child path will be identical to one forked before.

To identify the separation between the re-executed and the new parts of the path, Mousse uses a *forking skip depth* variable, which is offloaded alongside the concolic inputs when a path is offloaded. This variable specifies the number of symbolic forks to skip when re-executing the path from scratch. In other words, this variables splits the path into the re-executed and new parts using the number of symbolic predicates visited on the path.

Example. Figure 2.6 shows a simple hypothetical program. Assume that the analyst has marked the `arg` variable as symbolic. This means that three paths need to be executed as a result of two conditional statements on `arg` (lines 2 and 6). For simplicity of discussion, assume that Mousse is configured to execute one path at a time per device (i.e., no concurrency).

Mousse starts the execution and faces the first symbolic branch predicate at line 2. It forks the execution resulting in two different paths, and arbitrarily chooses to first execute the then-branch. This path issues a state-mutating ecall (line 3), which makes the other path (i.e., the else branch at line 5) inconsistent with the environment state. Mousse finishes

executing the then-branch path and then tries to resume the execution of the else-branch path. This path however needs to issue a state-revealing ecall (line 5) and hence cannot be executed in the device anymore. To offload this path, Mousse solves the path constraints (i.e., `arg < 13`) and finds a concolic input, e.g., `arg = 3`. It offloads this concolic input as well as the forking skip depth, which is 1 since the path has seen one symbolic fork so far.

Now imagine another device (or the same device after reboot) downloads this path to execute. To do so, it starts the execution from the beginning, marks `arg` as concolic, and assigns the concrete value of 3 to it. When it faces the first conditional with a symbolic predicate, it avoids forking due to the forking skip depth being 1. Mousse then inserts the concrete value of `arg` into the branch predicate and executes the `True` side of the branch (which is the else-branch). The importance of the forking skip depth is clear in this example: had the execution performed a fork here, the same path that was executed previously (i.e., line 3) would be executed again. The execution now resumes, forks another path at line 6, and manages to finish executing both paths. As can be seen, all three paths are eventually executed, one on the first device and the other two on the second device (or the second boot of the same device).

Global fork limiters. Loops create a problem for SSE and can result in a large number of program paths. Existing SSE solutions, such as S^2E, use fork limiters to limit the number of forks at a given program counter value. Mousse also uses fork limiters, but it needs to use a global one since the execution is distributed. To achieve this, Mousse's server implements global fork limiters. When performing a symbolic fork, each device contacts the server to inquire the value of the fork limiter, hence providing a global one. Moreover, Mousse uses both the program counter and the hash of the stack trace to identify a forking location. Compared to using the program counter only, this allows for a more accurate identification of loop forks. That is, this approach can differentiate between a function containing a loop being called from different call sites.

2.5.2 Environment-Forced Symbolic Variables

Outputs of ecalls marked as symbolic create a difficulty for offloading a path. Such variables are present when Mousse leverages its concretization with symbolic output or when we use a symbolic environment in our baseline experiments. We refer to these variables as environment-forced symbolic variables. In the presence of these variables, the solution to the path constraints might depend on specific values of these variables. However, these values cannot be simply fed to the program upon path re-execution.

To solve this problem, Mousse records and offloads some metadata information for each such symbolic variable. More specifically, it records the location of the ecall in the code (i.e., program counter value as well as the hash of the stack trace). When a device downloads this path to execute, it uses this metadata information to set the concolic value of the symbolic output accordingly. Along with the forking skip depth variable discussed earlier, this concolic value helps direct the path execution and avoid duplicate paths.

Note that we do not use the concrete value returned from the ecall itself for the concolic value of this variable on re-execution. This is because some paths cannot be triggered with the actual return value from the environment. If such a path is offloaded, a concrete value that can lead the execution correctly in this path needs to be offloaded as well.

2.6 Analysis

We have used Mousse to analyze Android I/O services. Specifically, we have performed three analyses: bug and vulnerability detection, taint analysis, and performance profiling.

2.6.1 Android I/O Services

We next provide some background information on Android I/O services. Android employs a large number of customized services tailored for each mobile device (more specifically, tailored for the hardware available in a specific mobile device). These services are often used to provide I/O API for applications. For example, the audio service is used to provide audio API while the camera service is used for camera API. Other such services include the WiFi service, bluetooth service, input service, sensor service, and telephony service.

An I/O service in Android may comprise of two components: a server component, which provides the application-facing API, and the Hardware Abstraction Layer (HAL), which provides the hardware-specific implementation needed to support the I/O functionality. The HAL service is implemented by the vendor of the hardware component and is typically closed source. In the rest of the chapter, we treat the server and HAL components as separate services and analyze them independently. This is because these two components are developed independently and even run in separate processes (especially in newer Android devices [77]). Smartphones incorporate a large number of closed source vendor services. For example, Pixel 3 incorporates 50 binary executables and 844 binary libraries for services from corresponding vendors, all adding up to 343 MBs of binary code.

2.6.2 Target Analyses

We next describe some of the analyses we perform using Mousse. Taking examples from S^2E [36, 37], we perform bug and vulnerability analysis and performance profiling. In addition, we perform taint analysis.

Bug and vulnerability detection. We develop checkers to analyze the execution of the program, both in symbolic and concrete modes, in order to find bugs and vulnerabilities.

First, we try to find *out-of-bounds access*, *null-pointer dereference*, *control-flow hijacking*, and *stack smashing* bugs and vulnerabilities. To do so, our checkers looks for symbolic memory accesses, i.e., when the memory address used is symbolic. Since in the analysis, we mark the inputs of the service API as symbolic, a symbolic address identifies a memory access that can be controlled by an attacker. We then check (using some manual effort) the constraints to see whether the access is adequately constrained. Second, we try to find *double-free* and *use-after-free* vulnerabilities. To do so, our checkers investigate the use of memory management APIs in `libc` including all heap allocation, reallocation, and deallocation calls, namely `free`, `malloc`, `calloc`, `realloc`, `memalign`, `posix_memalign`, `pvalloc`, `valloc`, and `aligned_alloc` to detect incorrect uses. Note that our checkers do have false positive reports requiring some manual effort in analyzing the reports. This is, however, a limitation of our checkers, not of Mousse.

Taint analysis. While Mousse can be used for different taint analysis goals, we deploy a specific analysis in this work: the flow of program inputs to its outputs. The results of this analysis can be used to enhance the accuracy of taint analysis for programs that use these APIs. For example, data flow analysis engines for Android apps (e.g., FlowDroid [17], Amandroid [114], and DroidSafe [57]) are unable to accurately model the data flow in Android APIs. Mapping the flow of the input to output of such API can complement these engines.

Performance profiling. Mousse can be used to profile the performance of different execution paths in a program. For example, given the cache properties (e.g., cache size, eviction algorithm, etc.), it can determine the number of cache misses in each program path. This can then be used to determine how some program inputs impact its performance and to find performance bottlenecks.

Testing methodology. Mousse can support arbitrary testing methods using SSE. However, in our evaluation, we focus on the following testing methods. The first method, which we mainly use to measure Mousse's performance, is *single-API testing*. By an API, we refer

to one of the procedures in the external interface provided by the program. Each I/O service in Android provides several procedures that can be called using IPC. For single-API testing, we initialize a service and then call a specific service API with symbolic inputs. Sometimes, when an API has a critical dependency on another API (e.g., all AudioProvider APIs require a call to `adev_open` first), we satisfy it in our test. The second method is *multi-API testing*. In one variant of this test, which we use mainly in our performance profiling, we first call one API with symbolic input and then call and execute another API concretely. In another variant, which we use mainly in bug and vulnerability detection, we may call multiple APIs, all (or some) with symbolic inputs. In the third method, which we also use for bug and vulnerability detection, we mark the variables read from the service configuration file as symbolic. We use this method to analyze the initialization code in the service.

2.7 Implementation

To implement the Mousse prototype, we developed 14,000 SLoC. In addition, we leveraged and integrated with Mousse parts of some existing systems, namely S^2E [36, 37], QEMU (user-mode execution) [21], and KLEE [27]. We use user-mode QEMU as the concrete execution engine in Mousse and KLEE as its symbolic execution engine. We use S^2E to integrate these two engines and to provide an extension framework to develop plugins (such as the checkers explained in §2.6.2). Mousse fully supports ARMv7 (which we use in our evaluations). We also plan to support x86 and ARMv8 in the future. The code that we developed is mainly for implementing process-level SSE (e.g., address space support, integration with user-mode QEMU, KLEE, etc.), support for ARM (both as the ISA of the program binary and as the ISA of the device to perform the analysis in), multi-threaded program support, environment-aware concurrency, distributed execution (including the server code), and the checkers described earlier. Note that using Mousse does not require any changes to the op-

erating system. However, in order to apply Mousse to Android I/O services, one needs root access on the smartphone.

Workflow. When Mousse is assigned to execute a program, the dynamic translator in QEMU first translates the program binary into Tiny Code Generator (TCG) [22] intermediate instructions. It then translates the TCG intermediate instructions into host instructions per basic block and starts the execution in concrete mode. In concrete mode, if it detects a symbolic variable, it switches to symbolic mode, translates the TCG instructions to LLVM instructions, and uses KLEE to execute the LLVM instructions. When no symbolic variable is present in a basic block, it resumes the execution back in concrete mode.

We adopted this workflow from S^2E, albeit with some differences. First, S^2E switches from symbolic mode to concrete mode when there are no symbolic values in CPU registers used in the next block. However, this approach is not feasible in Mousse because it cannot translate syscall handlers to instructions (since they are in the kernel). Therefore, it does not know if a syscall would access symbolic registers just based on the translated instructions. To solve this, Mousse adopts a more conservative approach. That is, it switches from symbolic mode to concrete mode only if all registers become concrete. Second, when facing a syscall, Mousse switches to native execution, whereas S^2E handles the syscall similar to the program's code.

State-mutating and state-revealing syscalls. In our experiments, we mark several syscalls as state-mutating including a driver `ioctl` syscalls and writes to a file, a socket, and a pipe. We also mark several syscalls as state-revealing including a driver `ioctl` syscalls and reads from a file, a socket, and a pipe. We note that we are conservative and assume all syscalls to a device driver can affect each other. It would be feasible to encode more fine-grained policies in Mousse, but that requires understanding the semantics of driver syscalls. Since ease of use is one of our goals, we opted for the easier, yet more conservative, approach.

Most ecalls are syscalls, e.g., an `ioctl` syscall to a device driver. However, another form of

ecall requires special attention: shared memory. For example, a program can use the `mmap` syscall to map, in its address space, the MMIO registers of a device or a memory buffer that is also accessed by a device driver. As another example, a program may use the shared memory support in the operating system to share a buffer with another process. Mousse treats writes and reads to/from such a shared memory segment similarly to explicit ecalls. We add support for various implementations of shared memory available in Android such as `mmap`, `ashmem`, and `ION`.

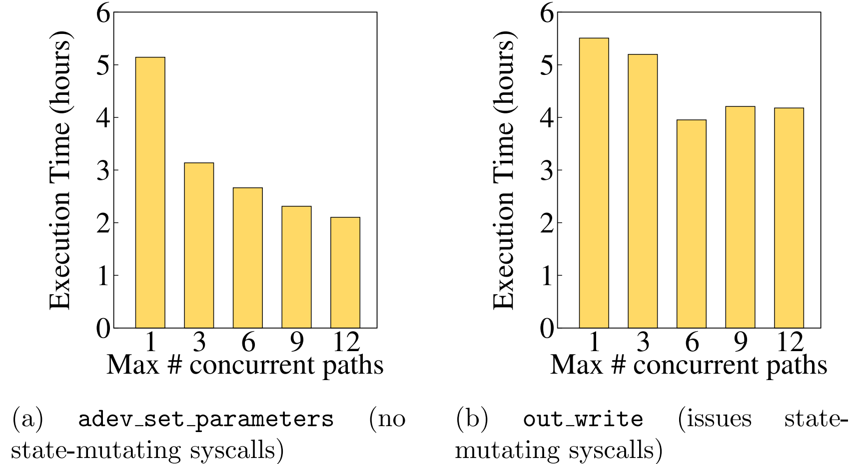
We do not currently support signals, as none of the programs we have analyzed use signals from the environment, e.g., from the driver. Instead, these programs use syscalls (such as `poll` and `select`) to receive notifications. We do, however, support per-process signals, such as `SIGTERM`.

2.8 Evaluation

We evaluate three aspects of Mousse: performance, code coverage, and analysis results. In our evaluation, we use five operating system services in three smartphones: two audio services in Pixel 3 (AudioServer and AudioProvider), two camera services in Nexus 5X (CameraServer and CameraDaemon), and the OpenGLES graphics libraries in Nexus 5. Unless otherwise stated, for distributed execution, we use five Pixel 3 smartphones, four Nexus 5X smartphones, and one Nexus 5 smartphone. We set the fork limiter threshold to 10 (similar to S^2E).

2.8.1 Performance

In this section, we provide empirical evidence that Mousse's solutions for environment-aware concurrency and distributed execution provide performance benefits. We also provide results



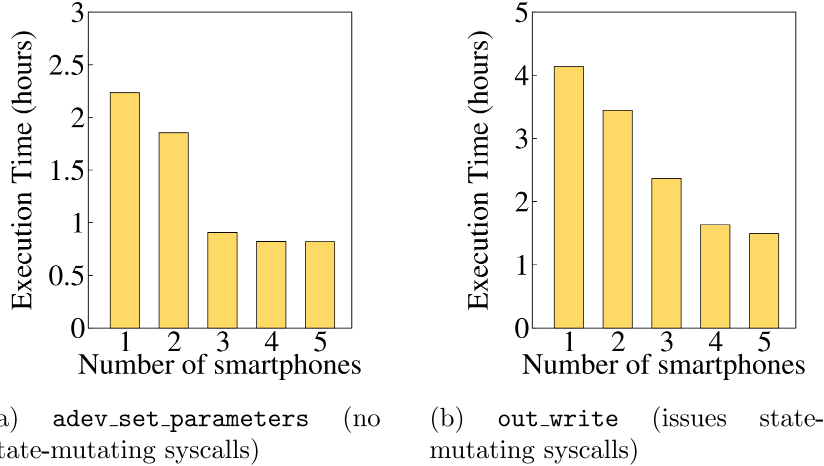
(a) adev_set_parameters (no state-mutating syscalls)

(b) out_write (issues state-mutating syscalls)

Figure 2.7: *Impact of environment-aware concurrency on execution time.*

quantifying the execution time of analysis using Mousse. We report the overall execution time of an experiment, from when it started until when the last path was executed. In the full paper [74], we also compare the performance of Mousse's process-level SSE design with an existing decoupled SSE solution. Note that we do not enable our checkers (§2.6.2) for these experiments so that we (*i*) we can measure the performance of SSE execution itself and (*ii*) we can compare the results with an existing SSE design, which does not have similar checkers. However, our measurements show that the checkers, if enabled, increase the execution time by 19.9%.

Environment-aware concurrency. Figure 2.7 shows the execution time of two APIs of the AudioProvider in Pixel 3 when varying the maximum number of concurrent paths allowed on the device. The figure shows significant benefit from concurrency for one API and modest benefit for the other. This is due to state-mutating syscalls. The first API (`adev_set_parameters`) does not issue state-mutating syscalls, allowing paths to execute concurrently with no restriction, resulting in 59% reduction in execution time. The second API (`out_write`) issues state-mutating syscalls, which limit concurrency (§2.4). However, even in this case, concurrent execution reduces the execution time by 24%. Moreover, for the second API, the figure shows an increase in execution time for 9 and 12 concurrent paths



(a) `adev_set_parameters` (no state-mutating syscalls)

(b) `out_write` (issues state-mutating syscalls)

Figure 2.8: *Impact of distributed execution on execution time.*

compared to 6. This is because when we increase the number of concurrent paths, there are more path execution conflicts (due to interactions with the environment) and hence more offloads. As mentioned earlier, an offloaded path is executed from scratch hence resulting in wasted execution time, which can negate the benefits of concurrency.

As discussed in §2.5, we empirically determine the maximum number of concurrent program paths. Accordingly to the results of this experiment, we set this threshold to 9 in the rest of the experiments.

Distributed execution. Figure 2.8 shows the execution time with distributed execution enabled. We show the results for using a different number of Pixel 3 smartphones (1 to 5). The results show that distributed execution significantly improves performance. Figure 2.8a shows the results for when there are no state-mutating syscalls. In this case, the performance improvement almost saturates with three devices, as all three devices can execute several paths concurrently. Figure 2.8b shows an API with state-mutating syscalls. In this case, adding the 4th and 5th devices further helps improve performance. Overall, distributed execution reduces the execution time by 63% and 64% for these two cases. Moreover, distributed execution and environment-aware concurrency together reduce the execution time by 84% and 73% for these cases.

Testing all APIs. To quantify the execution time of testing the APIs of a system service, we tested all the APIs of operating system services using the max number of devices available to us as reported earlier. Table 2.1 shows the results for three services. The table also shows the overall number of paths as well as the offloads due to environment consistency and due to resource constraint. The number of paths varies significantly depending on the API resulting in short (a few minutes) to long (a couple of hours) experiments. Also, the results show that both the environment and resource constraint may result in path offloads.

Service name	API name	Execution time (minutes)	# of path	# of offloads due to Res.	# of offloads due to Env.
GS	`eglCreateWindowSurface`	115.9	11	1	9
	`eglQuerySurface`	118.8	88	40	21
	`eglGetDisplay`	8.7	1	0	0
	`glCreateShader`	34.2	5	0	3
	`glShaderSource`	1605.8	371	148	95
	`glViewport`	14.6	6	5	0
AP	`adev_open_output_stream`	390.1	612	264	0
	`adev_open_input_stream`	170.1	566	234	0
	`adev_open`	2.2	12	0	0
	`adev_set_parameters`	107.7	237	122	0
	`adev_set_mode`	2.8	3	0	0
	`adev_set_voice_volume`	2.7	1	0	0
	`adev_set_mic_mute`	3.4	1	0	0
	`out_write`	89.6	50	24	10
	`out_set_parameters`	25.9	136	34	0
	`out_drain`	5.8	2	0	0
CS	`getNumberOfCameras`	47.6	46	28	3
	`connectDevice`	29.0	19	2	5
	`getCameraCharacteristics`	28.9	45	18	0
	`supportsCameraApi`	4.1	2	0	0
	`submitRequestList`	20.7	18	2	7
	`cancelRequest`	4.1	1	0	0
	`endConfigure`	4.2	1	0	0
	`createStream`	93.6	87	33	7
	`createDefaultRequest`	4.9	1	0	0

Table 2.1: *Single-API testing of operating system services with Mousse. Abbreviations used in the table: GS = GPU Stack, AP = AudioProvider, CS = CameraServer, Res. = Resource constraint, Env. = Environment consistency.*

2.8.2 Coverage

We measure the coverage of Mousse and compare it with that of concrete execution. We measure coverage in two steps: (*i*) the initialization coverage, i.e., the coverage resulting

from the initialization of the service and calling some other APIs that our API of interest has dependency on, and (*ii*) the API coverage, i.e., the added coverage when testing the API. Both Mousse and concrete execution result in the same coverage for the initialization phase. Hence, we mainly report the API coverage.

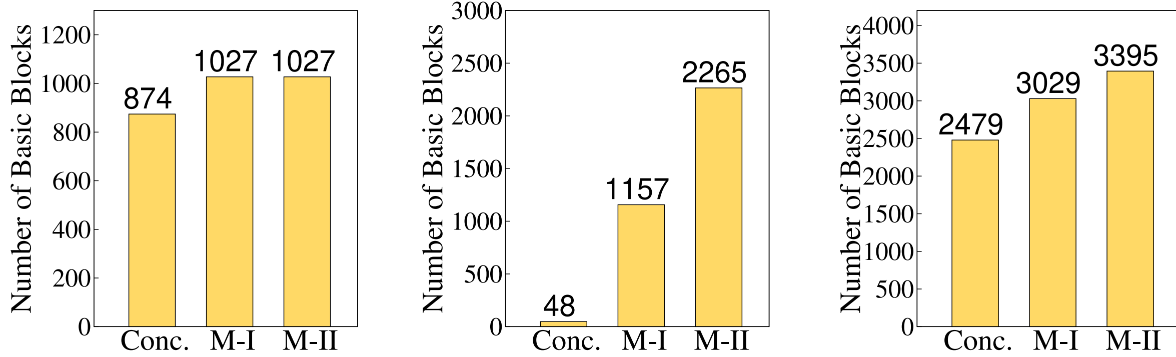
For concrete execution, we try two approaches and report the best one. One is using a known good input to the API that results in deep code coverage. The other is black-box fuzzing, where we try a large number of random inputs to the API and measure the combined coverage.

Figure 2.9 shows the API coverage for concrete execution and Mousse. It shows that Mousse achieves better coverage than concrete execution.

We also run these tests with a symbolic environment, for which we mark the output of a syscall as symbolic when the syscall is handled by the device driver used by a service, e.g., the audio device driver used by the audio service. In this case, as a result of path explosion, the three services that we test (i.e., CameraServer, CameraDaemon, and AudioProvider) all fail to correctly initialize (i.e., no paths within them successfully finish the initialization phase) even after 1 to 2 days of execution using Mousse's distributed execution with multiple smartphones.

2.8.3 Analysis Results

Bugs and vulnerabilities. We analyze all our services to find bugs and vulnerabilities. We find two new crash bugs (both null-pointer dereferences) and two double-free vulnerabilities. We then use Mousse to analyze these in the binary (demonstrating another benefit of Mousse, which can help analyze the execution). One null-pointer bug is due to accessing a gyroscope-related handle in the CameraDaemon without checking if it is null or not. The other is



(a) `adev_open_input_stream` of AudioProvider in Pixel 3

(b) `createStream` of CameraServer in Nexus 5X

(c) `on_mct_process_serv_msg` of CameraDaemon in Nexus 5X

Figure 2.9: *Code coverage for different APIs of Android I/O services. Conc. and M-II refer to concrete execution and Mousse with concretization II, respectively.*

due to access to a parameter buffer, which can be null. Moreover, one of the double-free vulnerabilities calls `free` on the same pointer three times.

Taint analysis. We analyze the propagation of inputs to the outputs of the AudioProvider service, which is a binary provided by the vendor. Our results show that no APIs propagate their inputs to their outputs with the exception of `out_write`, which returns its size input parameter as its output.

Performance profiling. We analyze the performance impact of audio quality configurations on the execution of audio playback code in the AudioProvider service. To do so, we configure the audio quality with symbolic inputs, call the playback API with concrete inputs, and then measure the cache misses. We model a two-level cache system using specifications from ARM Cortex-A53 (write-through LRU with 64 byte line size; 2-way associative/32 kB for L1D, 4-way associative/32kB for L1I, and 16-way associative/512 kB for L2).

Marking the audio quality configurations as symbolic results in 112 execution paths. We observe that different paths can experience 19% difference in the L1 data cache misses (i.e., the path with the maximum cache misses vs. the path with the minimum) whereas the cache misses for the L1 instruction cache and the L2 cache do not change noticeably. This shows

that different paths execute almost the same code, but with different data access patterns.

2.9 Other Related Work

Charm [95] ports some of the device drivers of mobile devices to run inside VMs. It does so by forwarding the drivers' I/O interactions with the hardware to the mobile device for execution. One may attempt to use Charm along with S^2E to analyze I/O services of mobile devices (indeed, this is the first approach we considered). However, Charm requires some engineering effort to support each device driver (in the order of days). Moreover, it may not port the drivers fully, e.g., it does not support DMA for a GPU driver. Finally, Charm does not virtualize the device, hence S^2E cannot use multiple VMs to interact with it. Since S^2E does not orchestrate interactions with an I/O device hardware (an untamed environment), concurrent execution of VMs would result in unexpected behavior.

Under-constrained symbolic execution, as mainly realized by Under-Constrained KLEE (UC-KLEE) [88, 89, 45], uses symbolic execution to analyze functions with systems code. It does not execute full program paths and simply considers the function arguments to be symbolic. This results in false positives. UC-KLEE therefore provides both automated heuristics and manual methods to add preconditions to the function's input in order to prevent some of the false positives. Mousse, on the other hand, can execute fully-constrained program paths.

DART and SAGE [52, 53, 54] automatically generate input for testing of programs by executing them, collecting path constraints, and solving the constraints, an approach otherwise known as concolic testing. Mousse also uses concolic inputs to drive the execution in a desired path (§2.5.1).

Mayhem [30] and Centaur [75] implement a decoupled SSE design. However, their designs are not conducive to analyzing programs with untamed environments. Mayhem runs the

concrete execution engine in a VM so that its state can be checkpointed. Hence, similar to S^{2}E, it requires to virtualize the hardware to analyze programs with untamed environments. Centaur uses a decoupled SSE design to analyze Android frameworks. However, it can analyze Java code only, whereas the programs of interest to us are typically written in native code. Moreover, it executes the initialization phase of the framework in concrete mode, and then moves to symbolic mode, after which it is not capable of switching back to the concrete mode.

AEG [19, 20] uses symbolic execution to automatically generate exploits. Driller [102] uses concolic execution to enhance the performance of fuzzing, an approach referred to as hybrid fuzzing. Both systems model the environment and hence cannot analyze programs with untamed environments.

Qsym [118] is a fast concolic execution engine used for hybrid fuzzing. Qsym avoids taking any path state snapshots and hence re-executes all the paths from scratch using concolic execution. As a result, it can allow the paths to interact with the actual underlying environment. However, Qsym does not provide support for environment-aware concurrency and needs the interactions with the environment to be side-effect free.

Cloud9 [38] distributes symbolic execution over multiple nodes. It, however, does not address the issue of the environment and targets pure symbolic execution (and not selective symbolic execution). Moreover, Cloud9's approach for distributing the execution is different from Mousse's. Cloud9 either uses state copying or state reconstruction to transfer a path from one node to another. State copying is not feasible for programs with untamed environments. State reconstruction is feasible and is indeed what Mousse does. However, Cloud9 uses a bitvector to encode the then/else decisions whereas Mousse uses concolic inputs. Moreover, Cloud9 does not deal with environmentally-forced symbolic variables.

Chipounov et al. define different execution consistency models for SSE, each resulting from

different transition points between symbolic and concrete executions and hence resulting in a different set of program paths being analyzed [37]. We note that concretization strategy I in Mousse results in the Strictly Consistent Unit-Level Execution (SC-UE) consistency model whereas concretization strategy II results in the Relaxed Local Consistency (RC-LC) model.

Chapter 3

Sifter: Protecting Security-Critical Kernel Modules in Android through Attack Surface Reduction

3.1 Introduction

The Linux kernel is an important part of the TCB of a mobile device using the Android OS. To compromise these devices (e.g., through privilege escalation), adversaries seek to exploit vulnerabilities in the kernel. Kernel exploits are very powerful; for example, a successful privilege escalation attack may allow malware to perform arbitrary operations. Kernel modules, e.g., device drivers, are especially concerning as they are often developed by third-party developers and contain many bugs (e.g., according to a report from Google on Android [103]).

However, not all vulnerabilities are created equal. More specifically, *those vulnerabilities that can be reached from untrusted programs pose a grave threat as they can be exploited by any program, e.g., any Android app.* Fortunately, Android makes most kernel modules inaccessi-

ble to untrusted programs, which greatly helps address this concern. For example, programs cannot directly interact with the camera and audio device drivers. They instead have to use these drivers through proxies, i.e., Android OS services with the unique permission to use these modules [1, 73].

This approach, while effective, cannot be applied to all kernel modules. This is because it impacts performance by adding another layer of indirection. Therefore, in Android, some performance-sensitive modules are still exposed to and directly used by untrusted programs. We refer to these modules as *security-critical* since their vulnerabilities are reachable by untrusted programs. Important examples of security-critical modules are Binder IPC, GPU device driver, and WiFi device driver. Indeed, an analysis from Google of bugs observed in Android's kernel shows that the majority of bugs reachable by untrusted program reside in these modules (excluding the `debugfs` and `perf` kernel modules as they can be simply disabled on production devices) [103].

Our goal in this chapter is to protect security-critical kernel modules in Android. Our idea to is to mitigate vulnerabilities in these modules by narrowing the attack surface through *system call (syscall) filtering*. Using syscalls is one of the key methods malware uses to exploit a vulnerability in these modules (§3.7). A syscall filter is deployed in the kernel. It analyzes every syscall issued by an untrusted process to these modules and decides whether to allow or reject it according to some policies. The hope is that, even though the vulnerability is still present, malware will not be able to exploit the vulnerability since it simply cannot trigger it using syscalls because of the filter.

We present Sifter, the first system designed to generate fine-grained, highly-selective filters for *mitigating vulnerabilities in security-critical kernel modules in Android*. Sifter is motivated by two simple observations. First, syscalls needed to interact with a kernel module follow a rich set of patterns, encoded in the module itself, often through a set of `ioctl` syscalls. Second, to exploit a vulnerability in a module, malware often issues syscalls with meticulously-crafted

arguments and sequences, which are not normally used by legitimate programs. Therefore, by rejecting bizarre-looking syscall patterns, Sifter can make it impossible for malware to trigger a large number of kernel vulnerabilities in security-critical modules even if the module is directly exposed to malware.

We pursue three goals in Sifter. First, Sifter filters need to perform deep inspections of syscall arguments, including direct arguments passed on registers as well as in-memory (sometimes nested) data structures. Second, they need to be stateful, considering the sequences of issued syscalls as opposed to making decisions on one syscall only. Third, the process of filter generation should be as easy as possible. More specifically, it should not require us to provide complex domain knowledge about the syscall interface of a kernel module.

To achieve these goals, Sifter leverages two recent advances related to Linux. First, it leverages eBPF programming, which is widely available in modern Android-based mobile devices, to securely deploy *stateful* syscall filters and to *safely access the program memory.* Second, Sifter leverages *syscall definition templates* available in syzkaller [11], the state-of-the-art kernel fuzzer, to provide the required domain knowledge for kernel modules.

We report a prototype of Sifter and use it to generate filters for two security-critical kernel modules used in many mobile devices: Qualcomm KGSL GPU device driver and Binder IPC.[1] We address two important practical challenges in our prototype. First, we show how we can deploy complex filters using eBPF by breaking them into multiple eBPF programs. Second, we devise solution to block collusion and Time-of-Check to Time-of-Use (TOCT-TOU) attacks.

We answer two important research questions:

Q1. Can effective filtering policies be automatically generated? Sifter automatically gen-

[1] We open source our prototype to facilitate future research and adoption at: https://trusslab.github.io/sifter/

erates and enforces three types of filtering policies. First, it restricts the syscall arguments depending on the type of argument, e.g., a length or a flag argument. Second, it serializes non-sleeping module-related *operations* (i.e., one or multiple syscalls used together to perform an operation). Third, it filters out legacy and deprecated operations never used by legitimate programs. We address several challenges in constructing these filters including dealing with argument outliers and identifying non-sleeping indivisible syscall sequences forming the aforementioned operations.

To demonstrate the benefits of these policies, we did a detailed study and evaluation of all CVEs reported for these two modules between 2016 and 2020 (41 in total). We show that Sifter can mitigate about half of all syscall-triggered vulnerabilities without *a priori* knowledge of the vulnerabilities (hence Sifter can mitigate zero-day vulnerabilities). More specifically, Sifter prevents the Proof-of-Concept (PoC) programs of these CVEs from triggering the vulnerabilities. Moreover, we discuss why the adversary cannot use mimicry attacks, which try to change the PoC to fool the filter.

In addition, using a large number of programs (and using disjoint sets of programs for training and testing), we demonstrate that Sifter filters do not break the execution of legitimate programs, resulting in 0% false positive rates, as long as an adequate number of programs are used for training.

Q2. Do such filters introduce prohibitive performance overhead? As mentioned earlier, these security-critical kernel modules are performance-sensitive (indeed, as discussed, that is why they are directly exposed to untrusted programs). We report a performance evaluation of Sifter. Our evaluation shows that Sifter adds a modest performance overhead to microbenchmarks. More importantly, it *incurs a very small or negligible performance overhead to real programs.* We also show that Sifter incurs a very small amount of energy consumption.

These results indicate that Sifter is a practical method to effectively curb the attack surface

```
1  long kgsl_ioctl_drawctxt_create(...) {
2    ...
3    write_lock(&device->context_lock);
4    idr_replace(&device->context_idr, context,
5      context->id);
6    write_unlock(&device->context_lock);
7    param->drawctxt_id = context->id;
8  done:
9    return result;
10 }
```

Figure 3.1: *Simplified code illustrating CVE-2017-9682.*

of security-critical kernel modules and mitigate many vulnerabilities.

3.2 Overview

Our goal in this work is to mitigate vulnerabilities in security-critical kernel modules in Android, i.e., those directly exposed to untrusted programs. Examples of such kernel modules are Binder IPC, GPU device driver, and WiFi device driver. Our key insights in this work are that (1) there are rich patterns in how legitimate programs issue syscalls to these kernel modules, and (2) in order to exploit a vulnerability in a module, malware uses syscall patterns different from those used by legitimate programs. Therefore, our key idea in Sifter is to develop filters in order to enforce the syscall patterns used by well-behaved, legitimate programs to interact with these modules.

3.2.1 Motivating Examples

CVE-2017-13162. To exploit this vulnerability, a malicious program must first issue an `mmap` syscall to Binder with a size that is smaller than `BINDER_MIN_ALLOC`. However, this argument value is never used by legitimate programs. This is because legitimate programs interact with Binder using the library `libbinder`, which passes a fixed size for `mmap`, one that is larger than `BINDER_MIN_ALLOC`.

CVE-2017-9682. Figure 3.1 shows this CVE, which is a Use-After-Free (UAF) vulnerability in the KGSL GPU driver. When a program creates a draw context by calling the `ioctl` syscall with the `IOCTL_KGSL_DRAWCTXT_CREATE` command, the driver first allocates and initializes the context. Then, at line 4, it adds the context to `context->idr`, where `idr` is a kernel API to build a mapping between integers (i.e., ID) and pointers, which in this case are pointers to contexts. Finally, after leaving the critical section protecting the `idr`, the driver retrieves the ID of the context, which will later be returned to the user. However, the final step at line 7 may lead to a UAF vulnerability if another thread is destroying the same context through another `ioctl`, `IOCTL_KGSL_DRAWCTXT_DESTROY`, making `context` a dangling pointer. In this example, to exploit the vulnerability, an adversary has to call the two `ioctl` syscalls from two threads concurrently. However, it makes no sense for legitimate programs to destroy a context simultaneously without knowing its ID returned by the creation. Indeed, during our training phase studying how legitimate programs interact with this driver (§3.4), the only syscall we see after `IOCTL_KGSL_DRAWCTXT_CREATE` is `IOCTL_KGSL_GPUOBJ_ALLOC`.

3.2.2 Goals

We pursue three goals in Sifter.

Goal I: perform deep inspection of syscall arguments. Syscall arguments are not limited to a few values passed on CPU registers. Many syscalls pass complex data structures in memory. For example, the third argument of the `ioctl` syscall is a pointer to a data structure, which itself might contain pointers to other data structures (i.e., a nested data structure). An example of such a syscall is `ioctl(fd, BINDER_WRITE_READ, ...)`, which programs use to pass a large set of "commands" to Binder.

Goal II: use stateful filtering policies. The same syscall (with the same argument) may

or may not trigger a vulnerability depending on the state of the kernel. Therefore, filtering policies that only inspect one syscall at a time to make a decision cannot cope with complex vulnerabilities. Therefore, an effective filter must be able to securely store and read some state information across syscall invocations and use the information for enforcing effective filtering policies.

Goal III: eliminate the required domain knowledge from the analyst. Since kernel modules are complex, requiring intimate knowledge of a module's syscall interface makes the process of generating fine-grained filters laborious and error-prone.

3.2.3 Feasibility

Is it even feasible today to deploy fine-grained syscall filters without impacting the performance and without requiring significant amount of domain knowledge (and hence manual effort)? We observe that this might be feasible only now due to two recent advances:

(1) Extended Berkeley Packet Filter (eBPF). Previously, an analyst who wanted to deploy a filter in Linux had two main options. One option was `ptrace`, which allows a tracer process to monitor another process (i.e., tracee process). Whenever the tracee calls syscalls, the tracer is woken up to check the tracee's syscall number and arguments. The major problem of `ptrace` is the performance overhead it introduces to the tracee process due to the context switches needed between the tracee and tracer on every syscall.

Another option was seccomp, which operates fully in the kernel and hence eliminates the aforementioned overhead. However, seccomp has two important limitations. First, it lacks the ability to inspect arguments passed in memory. Second, it cannot keep persistent state across syscalls, preventing stateful filters.

Fortunately, eBPF (extended BPF) support has been added to the Linux kernel (and hence

Android) recently [100] and it addresses the aforementioned limitations. eBPF has a richer (compared to BPF) set of instructions and supports a persistent data structure across syscalls (called *map*). In addition to the greatly extended capability, it also improves performance due to the use of just-in-time compilation. For enhanced security, eBPF programs are verified before being loaded into the kernel.

Indeed, there are several frameworks that can deploy eBPF programs in the kernel today. Since its introduction, *kernel tracing mechanisms* such as kprobe, tracepoint, perf event, and raw tracepoint have gradually supported eBPF [12]. In fact, our own tracer in Sifter leverages eBPF with tracepoint, as we will discuss. In 2018, integration of eBPF with seccomp was supported [43], although it has not been yet integrated into the mainline Linux. Finally, in 2020, integration of eBPF with Linux security hooks, also known as Kernel Runtime Security Instrumentation (KRSI), has been supported and added to the mainline Linux (starting version 5.7) [39, 97, 44].

Use of eBPF can enable us to achieve the first two goals in §3.2.2. Since all their policies are implemented in eBPF programs, Sifter filters can be deployed in practice using either seccomp/eBPF or KRSI. We use seccomp/eBPF in our prototype, but also provide a discussion of how Sifter can be deployed using KRSI (§3.9).

(2) Syscall definition templates from syzkaller. Kernel modules export a custom syscall interface, mostly implemented through the `ioctl` syscall. Each call to the syscall needs to specify a command number and pass a potentially complex data structure specific to that command. Developing effective kernel filters for a module hence requires understanding this interface, i.e., domain knowledge. Requiring the security analyst to provide this domain knowledge makes the process time-consuming and error-prone.

We observe that kernel fuzzers also require the same domain knowledge to be able to effectively fuzz a kernel module. They rely on *syscall definition templates* to achieve this goal.

As a result, security analysts have developed a large number of such templates. More specifically, syzkaller [11], the state-of-the-art kernel fuzzer provides such templates for 170 kernel modules (as of December 2021) including those specific to Android. Therefore, we reuse these templates in Sifter to eliminate the need for domain knowledge provided manually by the analyst. This enables us to achieve the last goal in §3.2.2.

syzkaller's templates include a comprehensive list of syscalls and the attributes of their arguments (e.g., field name, data type, data size, and the hierarchy of fields within arguments). The templates categorize the arguments into several types, for example: `IntType`, `LenType`, `FlagsType`, `ConstType`, `ArrayType`, `StructType`, `PtrType`, `BufferType`, `VmaType` (virtual memory areas), and `ResourceType` (values passed between syscalls, e.g., file descriptor).

We note that syzkaller might not have a template for a kernel module. In this case, the analyst needs to develop the template. This involves studying the syscall interface of a kernel module and then defining the syscalls along with their arguments using syzkaller's syscall description language. However, given the popularity of syzkaller and other kernel fuzzers [64, 4, 40, 94, 84, 65, 63, 116, 110], we believe that in the near future, many kernel modules will have their templates available. As an example, at the time of this work, syzkaller's main repository did not have a template for the KGSL GPU driver used in our prototype. However, our own research group had previously developed the template for this module as part of a fuzzing research project [104]. Therefore, we borrowed the template from that project and added minor improvements. This significantly reduced our manual effort.

3.2.4 Remaining Research Questions

Despite these new developments in Linux, two important research questions remain:

Q1. Can effective filtering policies be automatically generated? We show that we can automatically extract three different types of policies (i.e., argument limiting, operation serialization, and operation deprecation policies) related to a security-critical kernel module by observing how legitimate programs issue syscalls to the module (§3.4 and §3.5). We also show that our automatically generated policies can mitigate about half of all syscall-triggered vulnerabilities (§3.7). Moreover, we show that they do not introduce false positives (§3.8.1).

Q2. Do such filters add prohibitive performance overhead to legitimate programs? We demonstrate that our filters introduce very small or negligible performance overhead to real programs (§3.8.2). Moreover, they incur a very small amount of energy consumption (§3.8.3).

3.3 Threat Model

Our goal is to prevent malware from exploiting vulnerabilities in security-critical kernel modules through the syscall interface. We assume that malware controls one or multiple processes and can execute arbitrary code in them. We assume that malware can only leverage the syscall interface, but not other interfaces to the kernel. For example, we do not consider the hardware interface, e.g., incoming network packets and interrupts.

3.4 Workflow

Figure 3.2 shows the workflow of Sifter. There are three components in Sifter: the *automatic syscall tracer generator*, the *tracing agent*, and the *automatic filter generator*.

Imagine a security analyst who wants to use Sifter to generate a filter for a security-critical kernel module in Android. In the first step, the analyst uses the tracer generator to generate a tracer for that kernel module. To do so, the analyst feeds the syscall definition template of

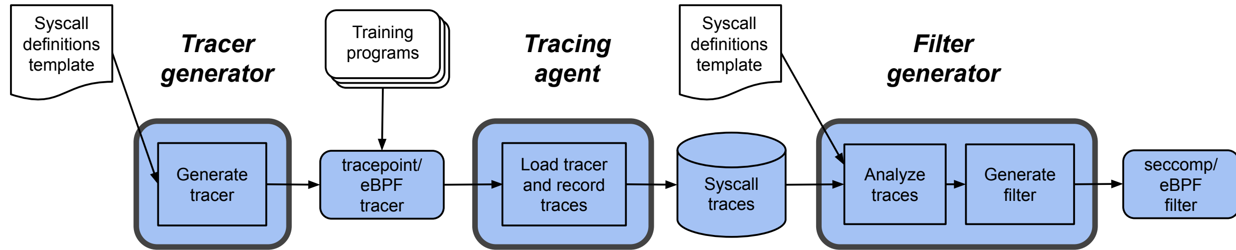


Figure 3.2: *Sifter's workflow. The blue shapes are part of Sifter and the white ones are external.*

that module (§3.2.3) to the tracer generator, which automatically generates the source code of a tracepoint/eBPF syscall tracer.

In the second step, the analyst uses the tracing agent to collect traces of syscalls from a set of *training programs*. The analyst needs to choose a representative and comprehensive set of training programs in order to capture all legitimate and expected syscall patterns (see Table 3.2 in §3.6.5 for the detailed list of the training programs in our prototype). To perform this step, the analyst uses the agent to deploy the tracer, execute the training programs, and collect the syscall traces. The tracing agent automatically hooks the tracepoint/eBPF tracer program to syscall entry and return points. The tracer logs the syscall number, the arguments in the registers and the timestamp for all syscalls into an eBPF map. When the syscall is for the kernel module of interest, the tracer will additionally perform a deep copy of arguments from the user space memory to additional eBPF maps.

In the third step, the analyst uses the filter generator to generate the filter. The generator reads the syscall traces, performs a series of analyses to extract filter policies (§3.5), and then generates a seccomp/eBPF filter to enforce the policies. The filter generator also uses the syscall definition template to generate code to parse syscall arguments at runtime.

In the fourth step, the analyst evaluates the false positive rate of the filter (§3.8.1). If not zero, the analyst chooses additional programs for training and goes back to the second step.

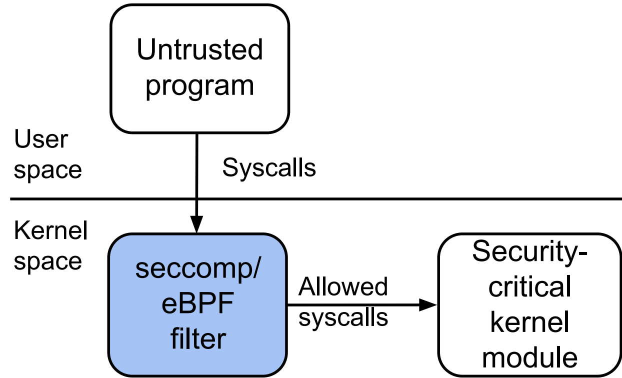


Figure 3.3: *The Sifter's filter limits syscalls of untrusted programs.*

This continues until the false positive rate reaches zero.

In the final step, the analyst deploys this filter into a target mobile device. As mentioned in §3.2.3, our prototype currently leverages seccomp to deploy the eBPF filter. In this case, the filter is deployed selectively for operating system processes. More specifically, similar to existing seccomp filters, the filter needs to be installed for the process before the execution of any untrusted code in that process. Given that a seccomp filter cannot be disabled once installed, the untrusted code cannot escape the filter. As also mentioned in §3.2.3, we can also use KRSI to deploy our filters. In that case, the filter will be deployed for a specific module (see §3.9 for a discussion on this matter). Figure 3.3 illustrates the filter in action.

We finally note that this workflow requires the analyst to invest time in selecting and running the training programs as well as inspecting and evaluating the generated filters. However, the analyst is not required to encode domain knowledge about the syscall interface of a kernel module, which would be a laborious and error-prone process.

Violation policy. When our filters detect a policy violation, they react according to a violation policy. Our default policy (which we also use in our prototype) is to block the violating syscall, potentially resulting in breaking/terminating the guilty program. This policy prevents exploitation of vulnerabilities and hence mitigates them. However, it is

possible to use a less strict policy, which just logs the violation and reports it for future auditing.

3.5 Filter Policies

Sifter generates three filtering policies based on the collected syscall traces from training programs. We next discusses these policies.

3.5.1 Argument Limiting Policy

This policy has stateless rules that constrain the values of syscall arguments of the following types: integers, length arguments, and flags. It also walks the arguments contained within arrays and structs and generate policies for them too.

A naive approach to implementing the argument policy is to blindly limit the arguments to only the values that have been recorded in the training phase. Unfortunately, this approach would require the filter to store a potentially prohibitively large number of values seen in the training phase, which is impractical. Moreover, it would require the training phase to cover extremely large number of training programs so that all possible values are seen. To address these challenges, Sifter generates the constraints statistically depending on their types. It does so in two steps: outlier analysis and policy generation.

Outlier analysis. Sifter prunes out *outlier* arguments from the traces. In our experience with a large amount of syscall traces, we have noticed that some legitimate programs issue erroneous syscalls, i.e., those with outlier arguments. For example, we have seen a binder command buffer containing an erroneous command with a 0x0 flag, which is undefined. To identify and remove outliers, Sifter analyzes the traces twice. In the first run, it builds

knowledge of the arguments. It then scans all the syscall traces again to drop the outliers.

Sifter prunes out the outliers based on shared characteristics of anomalies between the same type of arguments: for a *flag* argument, if the frequency of a unique value is below a given threshold and it only appears in one training program, then Sifter views it as an outlier. For a *length* argument, Sifter first computes its weighted mean, m, and the mean absolute deviation, d. Then, values that are $c_{\text{ol}} \cdot d$ away from m are considered as outliers.

We have found two outliers in all of the programs that we tested (§3.6.5) and we have manually confirmed that each of the detected outliers are indeed incorrect usages of a syscall.

Policy generation. Sifter analyzes the cleaned-up traces (i.e., traces without outliers) to generate argument policies. For *length* arguments, Sifter calculates the new weighted mean, m, and the mean absolute deviation, d. Then, it calculates the legitimate range of a length argument as $[m - c_{\text{d}} \cdot d, m + c_{\text{d}} \cdot d]$, or $[0, m + c_{\text{d}} \cdot d]$ if the lower bound is a negative number. For *flag* arguments, the policy will only allow the values seen in traces. Since the distributions of these arguments vary, we currently determine c_d empirically. However, we are considering automating this process by using the traces themselves to determine these constants.

Note that for pointers (i.e., `PtrType`), although the filter does not impose specific restrictions on the values, if the filter fails to copy the memory from the user space, the argument will be deemed invalid.

3.5.2 Operation Serialization Policy

A large number of vulnerabilities in the kernel are race conditions (e.g., 43% of all syscall-triggered vulnerabilities that we study in §3.7). To exploit a race condition, a malicious program often needs to issue syscalls simultaneously from multiple threads to cause a race. A straightforward approach to mitigating such vulnerabilities is serialization, which eliminates

concurrency.

A naive approach is to serialize each and all syscalls. However, this strawman approach has two important limitations:

First, it cannot prevent race conditions that are triggered as a result of a specific ordering of issued syscalls. For example, consider two syscalls issued consecutively, the first of which allocates a kernel structure containing a pointer and the second allocates the memory pointed by the pointer. If another syscall tries to dereference the pointer before the second syscall, a null pointer dereference will happen.

Second, it may cause a program to deadlock if the syscalls have some form of a dependency on each other. For example, we tested this strawman approach on the Binder module and noticed a deadlock in the few programs we tested (including a micro-benchmark and a game). This is because these programs use two threads to interact with Binder, one for sending and one for receiving IPC messages, where the latter sleeps in the kernel. Since these two threads have dependencies (i.e., the receiving thread would not receive any messages until the sending thread sends the message), serializing the syscalls causes a deadlock.

To address these limitations, we introduce the *operation serialization policy*. In this policy, we serialize *operations*, defined as a sequence of one or multiple syscalls always used together and in a specific order by programs to perform some higher-level operation related to the kernel module. Examples of these operations are Binder initialization, GPU initialization, and GPU cache flushing. Since these operations represent semantically-independent tasks, they are better candidates for serialization. We note that our filter will not be able to mitigate race conditions that rely on the order of execution of operations, but those vulnerabilities are by definition a subset of race conditions triggered by reordering the syscalls. Hence, our operation serialization policy is a stricter policy compared to the strawman syscall serialization policy.

It is important to clarify: since most of the operations on kernel modules are done through `ioctl` syscalls with different commands, we must differentiate these syscalls. This is achieved by treating `ioctl`s with different commands as different syscalls in the sequence. Besides, since `FlagsType` arguments might carry additional semantics in syscalls (e.g., an `ioctl`, `IOCTL_KGSL_SETPROPERTY`, may set different properties of the GPU driver depending on the `type` field in the structure, `kgsl_device_getproperty`), we also differentiate syscalls when these arguments are different.

Next, we discuss how we identify the operations by monitoring the syscalls issued by legitimate programs. We will also explain how we prevent deadlocks.

Operation detection. To detect operations, Sifter tries to find *indivisible sequences* of syscalls in the traces. The syscalls in such a sequence are always used by legitimate programs together and in a fixed order. In other words, one does not find a subset of an indivisible sequence in the traces.

Sifter identifies indivisible sequences in two steps. In the first step, it scans through syscall traces and breaks them down into smaller syscall sequences. Sifter breaks down the traces based on the temporal proximity of two kernel module syscalls defined by the time and the number of other syscalls between them. If the time of a kernel module syscall since the last kernel module syscall exceeds a threshold (250 ms in our prototype), it breaks down the sequence. (We determine this threshold, as well as a few other constants/thresholds discussed later, empirically by studying syscalls issued by programs.) During the scanning, there will also be other generic syscalls (i.e., syscalls not related to the kernel module of interest) between kernel module syscalls. As the number of these generic syscalls between two module syscalls grows, it is less likely the two module syscalls are associated and used together. Therefore, when the number exceeds a threshold (10 in our prototype), Sifter breaks the syscall sequence down.

However, all the sequences found in the first step are not indivisible as some might be a subsequence of others. The high level idea in the second step is to check if any of the sequences is a subsequence of another, in which case the longer sequence is split. This process continues until no new sequences are discovered.

After applying this algorithm to syscall traces from real programs (§3.6.5), we do find some interesting sequences, representing the aforementioned high-level operations used by programs. For example, we find a short `ioctl(fd, IOCTL_KGSL_GPUOBJ_ALLOC)`→`mmap(addr, PROT_READ|PROT_WRITE, MAP_SHARED, ...)` sequence, which suggest that the GPU object allocation operation always maps the GPU memory into user space after creating a memory entry in the GPU driver. We also find some longer operations. For example, we find an operation with 8 syscalls, all `IOCTL_KGSL_GETPROPERTY` but with different values for a *flag* argument, `type`: 0x13→0x17→0x18→0x15→0x19→0x20→0x1a→0x1b. This operation is used during the GPU initialization phase.

Deadlock prevention. As discussed earlier, serializing all operations can result in a deadlock. We observe that a deadlock happens only when one of the syscalls in an operation sleeps in the kernel (that is, this is a necessary but not a sufficient condition for the deadlock). Therefore, we do not serialize such operations. We identify these operations by measuring the execution time of their syscalls. We have only found two such operations (one in Binder and one in the GPU driver) and have confirmed (by analyzing the source code) that they do include sleeping syscalls.

3.5.3 Operation Deprecation Policy

Kernel modules often have several legacy and old syscalls not used by legitimate programs. However, these syscalls can be used by malware for exploits [69, 103]. We develop a policy that disallows such syscalls.

We identify deprecated syscalls by analyzing the syscall traces collected from legitimate programs. If a syscall has never been seen in the traces, our filter disallows using it.

However, our policy goes beyond disallowing single deprecated syscalls. It disallows deprecated operations. That is, if a sequence of syscalls has never been seen in the traces, our filter disallows it (even if all the syscalls forming the operation have been seen separately in different operations).

3.6 Implementation & Prototype

We implemented Sifter in Go and C++ in 6000 LoC. We developed the seccomp/eBPF filtering mechanism based on [43], and integrated it with the Linux kernel 4.9 for Android on Google Pixel 3. As discussed in §3.9, it is feasible to deploy Sifter's filters using the KRSI framework as well. We note that eBPF is widely available on modern Android-based mobile devices, i.e., those with Linux kernel version 4.9 or higher.

Using our prototype, we have generated filters for two security-critical kernel modules and tested them on Pixel 3: Binder IPC and Qualcomm KGSL GPU driver. We do not currently provide a filter for the WiFi driver (which is also a security-critical module) since our prototype currently only supports device files (e.g., /dev/binder for Binder IPC), but not sockets used by network drivers.

Table 3.1 provides some information about the two filters we have constructed. The table shows the number of rules of different policies of the generated filters. It also reports the number of eBPF instructions along with the total size of the maps (where we calculate the size of a map by multiplying the size of an entry by the number of entries).

We next discuss some important practical challenges that we needed to overcome to realize

	P_{Arg}	$P_{Op.S}$	$P_{Op.D}$	#Inst.	Map size (byte)
Binder IPC	15	3	5	1378	7700
GPU driver	60	23	16	1886	5734

Table 3.1: *Information of filters generated by Sifter. $P_{Arg}$is the number of argument limiting policies, $P_{Op.S}$is the number of operations in the operation serialization policy, and $P_{Op.D}$is the number of syscalls allowed by the operation deprecation policy. #Inst. is the number of eBPF instructions.*

Sifter.

3.6.1 Checking File Descriptors

Sifter enforces its policies per opened file descriptor. When a program needs to use a module, it first `open`s the module, which returns a file descriptor. All following syscalls are then issued by passing the file descriptor. We enforce the policies per file descriptor because kernel modules maintain different state information for each opened file descriptor.

This creates a challenge: the filter needs to know which module a file descriptor belongs to. This is challenging since seccomp checks syscall arguments at the syscall entry point right after switching to the kernel mode, but not at the implementation of the syscall handler. To solve this, we implement a new eBPF helper function, `bpf_check_fd`. The added helper function checks whether the `fd` argument of a syscall is for the targeted module or not.

3.6.2 Deploying Complex Filters using eBPF

A filter generated by Sifter is complex since it walks through deep memory arguments and maintains the state of the syscall sequences. In our first prototype, we implemented the filter in a single eBPF program. However, that exceeded the complexity limit of the eBPF verifier and thus was rejected. This was because the verifier could not traverse every possible path in the program to check its safety. To address this challenge, we break the filter into several

smaller filters: multiple argument limiting filters, one filter to reject single-syscall deprecated operations (and to reject syscalls not approved by at least one argument-limiting filter), and one filter to serialize the operations and to reject deprecated multi-syscall operations. These filters use an eBPF map to share information.

3.6.3 Preventing Collusion Attacks

When analyzing syscall traces to extract operations (§3.5.2), we look at syscalls triggered by each thread separately. This is because in a well-behaved program, there is no reason to use more than one thread to issue a set of dependent syscalls to realize an operation together. However, when enforcing the operations (i.e., sequence of syscalls), we cannot simply enforce them per thread. If we do, malware can use colluding threads or processes to bypass the filter. We devise solutions to prevent such collusions, as discussed next.

First consider the case of colluding threads. This case is taken care of by serializing the operation issued on a specific file descriptor. To implement the critical section, we add three eBPF helper functions, `bpf_lock_fd`, `bpf_unlock_fd` and `bpf_wait_if_fd_locked`. Note that our helper functions lock the underlying `struct file` of the file descriptor, as opposed to the file descriptor itself.

Now consider the case of colluding processes. Malware can achieve this in two ways. One way is to fork a child process and use the threads in this process to break the serialization. We prevent this case by sharing the aforementioned lock with child processes. Another way is to use two independent processes. In this case, one process needs to be able to `dup` the file descriptor for the other process. Therefore, we do not allow `dup` to duplicate the file descriptors of our security-critical kernel modules.

3.6.4 Preventing TOCTOU Attacks

TOCTTOU attack is a check-evading technique, which succeeds when the target of check can be modified after check and before use. To prevent TOCCTOU attacks on in-memory arguments, Sifter caches the arguments by their addresses when being examined by the filter. Later when the kernel module fetches the argument by calling `copy_from_user` and `get_user`, instead of copying the argument from the user space again, Sifter returns the cached value. Therefore, even if a malicious program uses another thread to modify the argument, the modified value will not be consumed by the module.

We also prevent TOCTOU for file descriptors. Since there is a window between the filter and the actual syscall handler, the file pointed to by the file descriptor might be changed maliciously to bypass the filter. The attacker can mount an attack by first calling the syscall consisting illegal arguments to a file other than the kernel module of interest. Then, right after it passes the filter, the attacker can "redirect" the call by closing the file pointed by the original file descriptor and then duplicating the file descriptor of the kernel module to take the place of the original file descriptor in the argument. However, as discussed, we do not allow `dup` to duplicate on the file descriptors of our security-critical kernel modules, which blocks this type of TOCTTOU attack.

3.6.5 Training Programs

We used a diverse set of training programs for our filters. For Binder IPC and the GPU driver, we record syscalls issued by 60 Android apps downloaded from the Google Play store. The training programs are selected from the top charts of various categories based on popularity and review score. They include popular video streaming apps, shopping apps, games, social media apps, and tools. Table 3.2 shows the detailed list of the training programs. For each of them, we start the tracing agent and manually interact with the app for 30 minutes

with the goal of exercising as much functionality as possible. The syscall tracing results are later used to generate filters. This manual interaction with the app is currently the most time-consuming aspect of our system. However, we think the manual effort is acceptable since there are only a limited number of security-critical kernel modules that programs can directly access in Android. Besides, the two security-critical kernel modules we analyzed are widely available: Binder is available on all Android devices and Qualcomm GPUs have a significant market share. Therefore, the effort could benefit many devices in the wild with the potential of mitigating zero days. One option to automate the manual trace collection is to use a monkey, e.g., the Android UI/Application exerciser [5]. However, a monkey might not be able to invoke as many functionalities of the app as we manually do since it often gets stuck at certain UI elements. We think that tuning a monkey to generate syscall traces with high coverage needed by Sifter will be a useful future work.

Apps	Games	Google
Zoom, Snapchat, Discord, Webtoon, Duolingo, McDonalds, Doordash, MyFitnessPal, AllTrails, Spotify, Shazam, Opera News, Adobe Acrobat, Microsoft Word, Amazon Shopping, Walmart, Facebook, Twitter, Reddit, NFL, Booking.com, The Weather Channel, Twitch, Pintrest, eBay, Wish, Dropbox	Genshin Impact, Roblox, Terraria, Dragon Ball Z Dokkan Battle, Tomb of the Mask, Crossy Road, Subway Surfers, Angry Bird Friends, Hungry Shark World, Block Craft 3D, Candy Crush Saga, Coin Master, My Talking Tom 2, Best Fiends, Toonblast, Cut the Rope 2, Need for Speed: No Limits, Design Home: Real Home Decor, Fishing Clash, SimCity Buildit, 8 Ball Pool, Clash of Clans, Plants vs. Zombies, Egg Inc, Snake.io	Youtube, Gmail, Maps, Photos, Calendar, Translate, Sheets, Earth

Table 3.2: *Training programs used for generating syscall policies.*

3.7 Effectiveness Study

We present a study of 41 Linux CVEs from the Binder IPC module and the Qualcomm KGSL GPU device driver used in many mobile devices. These CVEs are all that we found for the period of 2016-2020. We collect these CVEs from the MITRE CVE database and the

Android security bulletin. We have two key goals in this study. First, we use the study to show empirically that a large number of vulnerabilities are triggered by syscalls and hence can, at least theoretically, be mitigated by syscall filters. Second, we use the study as a framework to evaluate the effectiveness of Sifter.

Components	V_{sifter}			$V_{syscall}$- V_{sifter}	V_{all}- $V_{syscall}$
	Argument limiting	Op. serialization	Op. deprecation		
Binder	19-2181^{2}, 17-13162^{2}		20-0030^{3}, 19-2215^{4}, 19-2000^{4}	20-0041^{5}, 19-2214^{2}, 19-2213^{3}, 19-2025^{3}, 19-1999^{3}, 18-9465^{3}, 17-17770^{5}, 17-13164^{1}, 16-6689^{1}	18-20510, 18-20509, 16-8402, 16-6683
GPU driver	18-5831^{4}, 17-7366^{5}, 16-2468^{2}	18-13905^{3}, 17-9682^{3}, 17-8262^{3}, 16-8479^{3}, 16-2504^{3}, 16-2503^{3},	17-15829^{3}, 17-15820^{3}, 17-14886^{5}, 16-2602^{2}	19-10571^{2}, 19-10567^{5}, 19-10529^{4}, 17-14891^{1}, 17-11092^{3}, 17-11044^{4}, 16-6749^{1}, 16-2067^{5}	19-10545, 18-3571
Total # bugs	5	6	7	17	6

Table 3.3: *Analysis of CVEs in Binder and Qualcomm KGSL GPU driver (CVE-20YY-XXXX is shortened to YY-XXXX^{T}, where T is the vulnerability type – 1: information leakage, 2: OOB access, 3: data race, 4: kernel API misuse, and 5: logical bug).*

Out of all the vulnerabilities (V_{all}), we determine those that are triggered by the use of syscalls ($V_{syscall}$). We find that 35 of the vulnerabilities (85%) fall in $V_{syscall}$. Out of the 6 remaining vulnerabilities, five are triggered through the *debugfs* and one through the initialization path with an erroneous configuration. These vulnerabilities are out of the scope of a syscall filtering solution.

We then determine whether Sifter can mitigate the syscall-triggered vulnerabilities ($V_{syscall}$). For each such vulnerability, we first determine the sequences of syscalls needed to trigger it. This is done mainly by analyzing the source code and studying PoC programs. We assess the effectiveness of Sifter by seeing if the syscall sequence to trigger the vulnerability violates the policies generated by Sifter (in which case Sifter mitigates that vulnerability). If we determine that Sifter can mitigate a vulnerability, we confirm it by actually running the PoC and checking that the Sifter's filter neutralizes it. Whenever PoCs were not available, we developed them ourselves, a process that required a significant amount of effort. We have released the 12 PoCs that we have developed alongside the source code for Sifter, so that

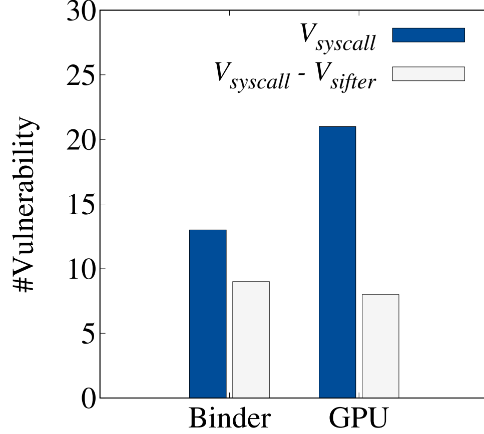


Figure 3.4: *The numbers of triggerable vulnerabilities in $V_{syscall}$ before and after applying filters.*

our results could be reproduced.

The number of triggerable vulnerabilities in $V_{syscall}$ before and after applying filters is illustrated in Figure 3.4 and the detail breakdown is shown in Table 3.3. Our analysis shows that 18 of the vulnerabilities are mitigated using Sifter (V_{sifter}). They account for about 51% of syscall-triggered vulnerabilities ($V_{syscall}$).

We further analyze the severity of the vulnerabilities. Our analysis shows that the vulnerabilities mitigated by Sifter have high severity, i.e., their average CVSS rating is 7.3. This shows the effectiveness of Sifter in mitigating serious vulnerabilities.

Effectiveness of different policies. We find that different policies have varying levels of importance in mitigating different types of vulnerabilities. We discuss how they succeed below.

First, the argument limiting policy mostly mitigates Out-Of-Bounds (OOB) vulnerabilities. Figure 3.5 shows one such vulnerability, CVE-2016-2468, in the Qualcomm GPU driver. During GPU memory allocation, the driver updates `sglen` in the loop at line 12 based on `len`, which gets directly assigned from a user-controllable argument of the function, `size`, earlier at line 8. Since the datatype of `len` is a signed integer, assigning it with a value greater

```
1   static int
2   _kgsl_sharedmem_page_alloc(struct kgsl_memdesc *memdesc,
3                              struct kgsl_pagetable *pagetable,
4                              size_t size) {
5     int ret = 0;
6     int len, page_size, sglen_alloc, sglen = 0;
7     ...
8     len = size;
9     ...
10    while (len > 0) {
11      ...
12      sg_set_page(&memdesc->sg[sglen++], page, ... );
13      ...
14    }
15    ...
16    sg_mark_end(&memdesc->sg[sglen - 1]);
```

Figure 3.5: *Simplified code illustrating CVE-2016-2468.*

than 0x80000000 will result in a negative `len`. As as result, `sglen` never gets updated in the loop, and when being used to index an array at line 16, OOB access happens. However, during the training, it never exceeds 0x80000000. The argument policy generated by Sifter limits its upper bound to around 0x1E000000, therefore preventing it from being exploited.

Second, the operation serialization policy in our analysis mostly mitigates race condition vulnerabilities, especially UAFs. We explain how it works using CVE-2018-13905 as an example, which is illustrated in Figure 3.6. In the Qualcomm GPU driver, the book-keeping of sync-sources is done by using `idr`. The user can destroy a sync-source by calling the `ioctl` syscall with an argument specifying the ID. The driver first invokes `kgsl_ioctl_syncsource_destroy` at line 13. It then looks up the sync-source from the `idr`, calls `kgsl_syncsource_cleanup` to remove it from the `idr`, and then releases the sync-source. While every operation on `idr`, namely `idr_find` and `idr_remove`, seems to be protected by a `spinlock`, the destroy process is not atomic. When one thread is trying to destroy a sync-source, after it finds the sync-source and before removing it, another thread might acquire a reference to the same sync-source. After it is destroyed by the first thread, access to the dangling pointer in the second thread will cause a UAF.

To exploit this vulnerability, an attacker will need to issue the racy syscall from two con-

```
1  static void kgsl_syncsource_cleanup(struct kgsl_process_private *private,
2                                      struct kgsl_syncsource *syncsource) {
3     ...
4     spin_lock(&private->syncsource_lock);
5     if (syncsource->id != 0) {
6        idr_remove(&private->syncsource_idr, syncsource->id);
7        syncsource->id = 0;
8     }
9     spin_unlock(&private->syncsource_lock);
10    ...
11 }
12
13 kgsl_ioctl_syncsource_destroy(struct kgsl_device_private *dev_priv,
14                               unsigned int cmd, void *data) {
15    ...
16    spin_lock(&private->syncsource_lock);
17    syncsource = idr_find(&private->syncsource_idr,
18                param->id);
19    spin_unlock(&private->syncsource_lock);
20    ...
21    if (syncsource == NULL)
22       return -EINVAL;
23    kgsl_syncsource_cleanup(private, syncsource);
24    return 0;
25 }
```

Figure 3.6: *Simplified code illustrating CVE-2018-13905.*

current threads. In this case, it is `IOCTL_KGSL_SYNCSOURCE_DESTROY`. However, during automatic syscall analysis running training programs, we discover that the syscall only appears in one operation (i.e., an indivisible sequence of syscalls), `IOCTL_KGSL_SYNSOURCE_CREATE`→ `IOCTL_KGSL_SYNCSOURCE_DESTROY`. When this syscall sequence is enforced by the filter, during a call to the syscall sequence, no other threads would be allowed to make syscalls to the driver and therefore the vulnerability becomes impossible to exploit.

Finally, our operation deprecation policy is effective against attacks exploiting vulnerabilities in deprecated syscalls. An example is a race condition bug, CVE-2017-15829, in the deprecated `IOCTL_KGSL_SPARSE_BIND` syscall.

Non-mitigated vulnerabilities. An example of these non-mitigated vulnerabilities is CVE-2020-0041 in the Binder IPC. The vulnerability is caused by an incorrectly calculated variable used for checking the boundary. This allows an attacker to violate the rules of how binder objects should be placed in the binder transaction buffer. The binder transaction buffer may hold a series of objects with hierarchical relationships and requires specific logic

to validate its correctness. Sifter unfortunately cannot mitigate this vulnerability.

Another example of a non-mitigated vulnerability is CVE-2017-14891 in the GPU driver. The driver does not zero out a variable in the beginning, causing it to leak sensitive stack data to the user. Since the filter does not have control of the leaking path, it fails to mitigate this vulnerability.

The third example, CVE-2019-2213, is a race condition vulnerability in the Binder IPC. It cannot be mitigated by Sifter because not only the racing paths can be triggered with normal arguments, but also concurrent access to the shared data cannot be serialized by the operation serialization policy. More specifically, since the `ioctl` with command `BINDER_WRITE_READ` may sleep during invocation, the kernel module cannot be locked as doing so will break it. As a result, this vulnerability cannot be mitigated by the filter.

3.7.1 Mimicry Attacks

One might wonder about Sifter's defense against mimicry attacks [109]. That is, one might wonder whether Sifter's filters can be bypassed by (1) crafting the syscall arguments, (2) substituting some syscalls in the syscall sequence with equivalent syscalls, or (3) inserting no-op syscalls or delays between the syscalls in an operation to deceive the filter. (These are all types of mimicry attacks that we could think of.) In our analysis, Sifter is secure against such mimicry attacks, as discussed next.

(1): unlike existing syscall filtering mechanisms that only check arguments in registers, Sifter checks deep in-memory arguments. Therefore, if syscall arguments are maliciously modified, they will be detected by Sifter. We do note that Sifter might not check all the arguments of a syscall. This can happen if the syscall descriptions of the module borrowed from syzkaller are not complete. But according to our analysis, our argument checks are adequate to thwart

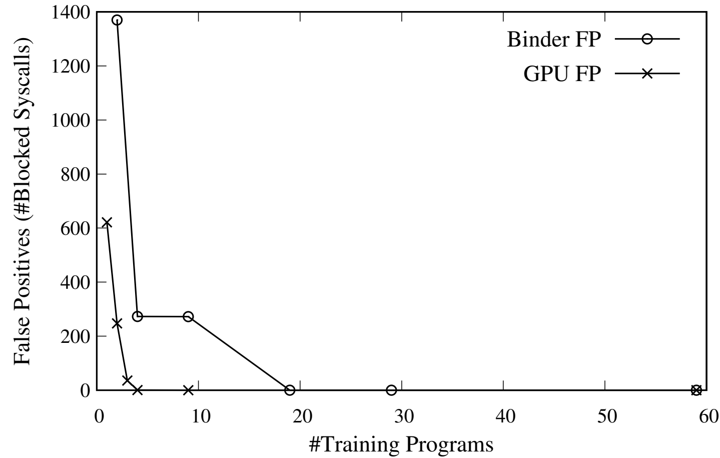


Figure 3.7: *False positives of filters trained and tested using different numbers of training programs.*

such mimicry attacks for all the CVEs we analyzed.

(2): in a complex kernel module such as the GPU driver, it is possible to have multiple entry points that can reach a vulnerability (i.e., equivalent syscalls). However, in our analysis, these equivalent syscalls are mostly legacy ones, hence our operation deprecation policy rejects them. According to our analysis, no such mimicry attacks are feasible for all the CVEs we analyzed.

(3): our filters ignore delays and no-op syscalls when detecting syscall sequences of operations, hence these attacks will not be effective.

3.8 Evaluation

3.8.1 False Positive Analysis

Sifter's dynamic approach in generating syscall policies could yield stricter filters when compared to statically analyzing syscall usages of the kernel module from applications or libraries.

However, on the other hand, it has a higher chance of accidentally blocking or reporting (depending on the violation policy discussed in §3.4) a legitimate syscall, i.e., a false positive. In this section, we demonstrate that, with enough data, our filters can achieve a 0% false positive rate.

We use a leave-one-out cross-validation method to assess how filters trained with different sizes of data set (numbers of traces) perform. We first randomly select different numbers of programs from all the available program traces collected from real programs. Next, in each data set, we leave one program out for testing and use the rest as training data. Then, we feed the traces of the training programs to Sifter's filter generator to produce syscall policies.

Finally, we configure the filter generator into testing mode and feed the testing traces. The filter generator then acts as the syscall filter imposing the policies and reports syscalls that violates the policies when it scans through the traces. Note that during training, we use pre-determined thresholds for each type of argument and policy, and do not change them. Besides, to prevent overfitting the model, the cross-validation method uses the trace unseen by the trained model for testing. In addition, the train-and-test process is repeated ten times independently without carrying over the result for each size of the data set or across iterations.

The result, illustrated in Figure 3.7, shows that initially, filters of both modules have significant amount of false positives, which are contributed by all policies due to the lack of training data. However the false positive rate is reduced dramatically with increasing the training data set size. For GPU driver, we can reach 0% false positives after using 9 programs for training. For Binder, it needs more training data to reach 0% false positive. The false positives drop to 0% after we increase the number of training programs to 19. Note that the required amount of programs for different kernel modules may vary, and the train-and-test method can serve as a indication of whether the training data is enough.

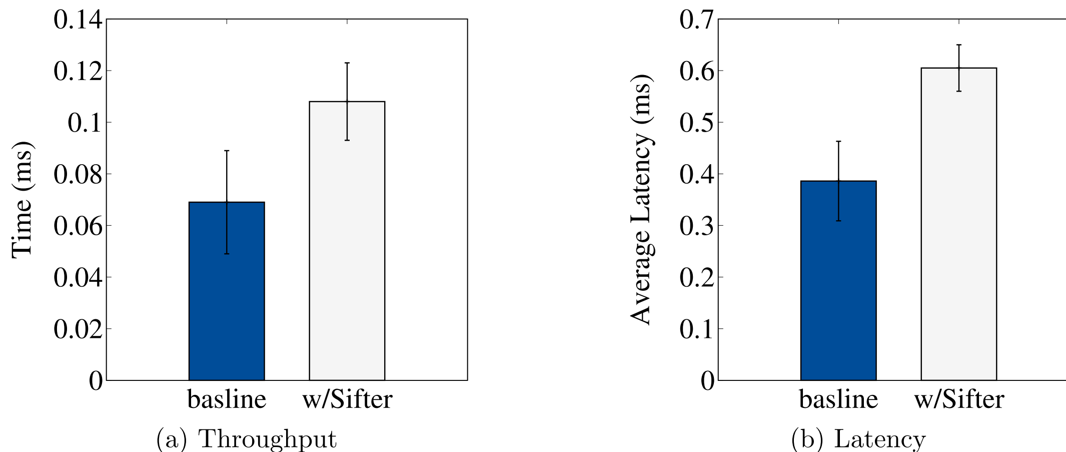
Besides, while it is equally important to reduce false positives for all policies as a single false positive can potentially break a normal program, a false positive in operation serialization policy could be more problematic. An overly long syscall sequence that is not truly indivisible may lock the kernel module and prevent other threads from using it, causing the program to freeze.

Our use of syscall traces collected from real programs provides a reproducible way to evaluate the false positive rate under different training set sizes. Since we record all the syscalls issued by programs, this evaluation methodology does not add any inaccuracies. To further demonstrate that our filters can achieve 0% false positive rates, in the second step, we also apply both of our filters to a couple of programs (Booking.com, Youtube, Terraria, and Subway Surfers) running in a smartphone and verify that the filter does not break normal usages of these programs.

It is worth mentioning that although we do not change the aforementioned thresholds in our evaluation for both modules, they theoretically can differ for kernel modules even for the same type of arguments as they carry different semantics. For the threshold used in operation serialization policy, it serves as a hint to break down the syscall sequence. Therefore, it should be set rather large than small to avoid breaking an indivisible sequence. A large threshold will only cause the indivisible sequence extraction process longer. For the length-type arguments in the argument limiting policy, the threshold is determined by mean absolute deviation. Therefore, if the distribution of arguments changes a lot, it could require more attention from the analyst.

3.8.2 Performance

We measure the performance overhead of Sifter on Binder IPC and the Qualcomm KGSL GPU driver. To evaluate the performance overhead introduced by Sifter, we install the filters



(a) Throughput

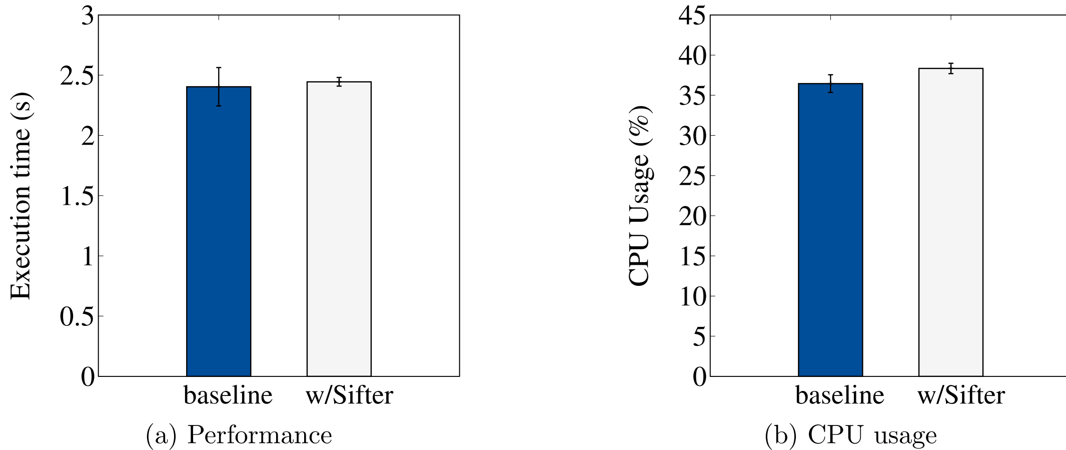
(b) Latency

Figure 3.8: *Performance overhead of Sifter on Binder micro-benchmarks.*

generated by Sifter on selected benchmark programs including both synthetic benchmarks (i.e., micro-benchmarks) and real programs (i.e., macro-benchmarks). Then, we report the execution time as well as the CPU usage. We run each experiment ten times and show the mean and standard deviation (the latter using error bars).

Binder. To evaluate the impact of Sifter on Binder, we choose two micro-benchmarks, `BinderThroughput` and `schd-dbg`, and one macro-benchmark, `SurfaceFlingerStress`. `BinderTroughput` measures the average time consumed by Binder transactions of various sizes, and `schd-dbg` measures the average latency of Binder transactions. The results of the micro-benchmarks, shown in Figure 3.8, show that Sifter increases the transaction time by 49%. The latency micro-benchmark shows transactions on average spend 56% more time in the kernel.

However, the overhead observed in micro-benchmarks might not translate directly to degraded user experience since Binder IPC normally takes only a small portion of a program's execution time. Therefore, it is also important to understand their real-world impact on programs by running macro-benchmarks. We choose the macro-benchmark because it has high Binder utilization, i.e., 50% of its syscalls are Binder `ioctl`s. Figure 3.9a shows the execution time of `SurfaceFlingerStress`, which is a test program stressing `SurfaceFlinger`



(a) Performance

(b) CPU usage

Figure 3.9: *Performance overhead of Sifter on Binder macro-benchmark (SurfaceFlinger-Stress).*

by creating and destroying surfaces for 1,000 times in 10 threads, respectively. Therefore, we use it to simulate a Binder IPC intensive application. Although Sifter introduces a noticeable overhead to Binder transactions, it only slows down the extremely IPC-intensive program by 1.7% and increases the CPU usage by 5% (Figure 3.9b).

GPU driver. To evaluate the performance overhead of the filter on GPU in real world scenarios, we apply the filter to three 3D benchmarks. We measure the average CPU usages and the frames-per-second (FPS) during the testing session. Figure 3.10a shows the FPS of the three macro-benchmarks. The performance overhead introduced by the syscall filter in terms of FPS is almost negligible whether the benchmark programs stress the GPU or not. The differences of FPS are within the measurements' error margins. Figure 3.10b shows the CPU usages of the benchmarks before and after installing the filter. The CPU usage increase is also almost unnoticeable in the first GPU benchmark. In the second and third benchmarks, the CPU usage of the program with filter is slightly lower than the one without by 4.4% and 3.3%, respectively. (We suspect that the small amount of idle time caused by the locks in our serialization policy results in the slightly lower CPU usage when the filter is used.) This again shows that the performance overhead introduced by Sifter is almost negligible in real world scenarios.

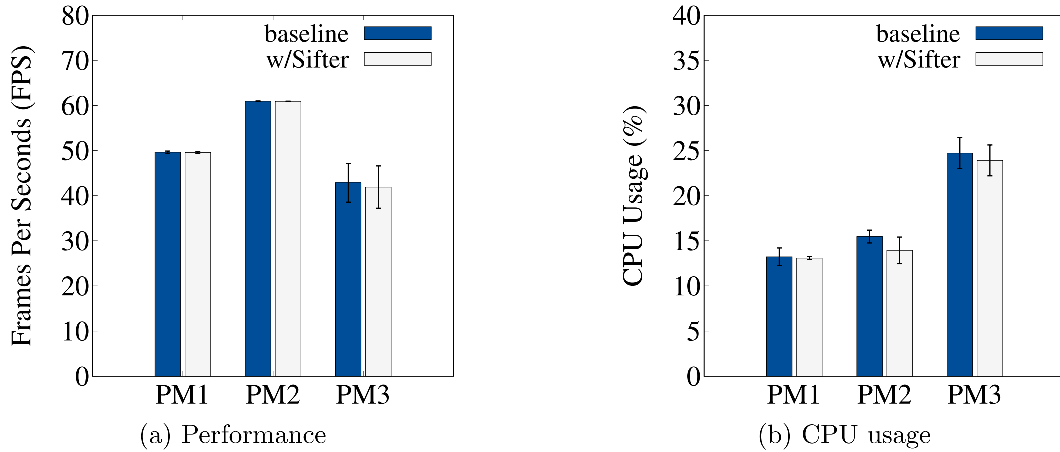


Figure 3.10: *Performance overhead of Sifter on GPU macro-benchmarks (PM1: Passmark 3D simple, PM2: Passmark 3D complex, PM3: Passmark OpenGL ES).*

Scalability. To evaluate the performance impact of multiple filters being installed to a program, we apply different combinations of filters to a 3D game, Subway Surfers, that use both Binder IPC and the GPU. The FPS and CPU usages difference are all within the error margin even when both Binder and GPU filters are installed.

3.8.3 Energy Consumption

To evaluate the energy consumption introduced by the filters, we installed the filters for both Binder IPC and GPU driver to the aforementioned 3D game. Then we measure the energy consumption of a six-minute game play for ten times. Compared to running the program without filters, the energy consumption is increased by 0.73mAh (which is 0.025% of the total battery capacity).

3.8.4 Training Time

The training time of a filter is proportional to the training data set size. When using the largest data set, which comprises 60 30-minute traces with a total size of 131 GB, it takes

Sifter two hours to analyze the traces and generate a syscall filter on an Intel Xeon CPU E5-2697v4 machine using a single core.

3.9 Discussions

Deploying Sifter filters with KRSI. In general, it is possible to deploy Sifter filters with KRSI. To do so, a mapping of syscalls to Linux security hooks is required since seccomp filters and KRSI eBPF programs are invoked at different locations. While seccomp filters hook to the syscall entry, KRSI eBPF programs are called at Linux security hooks placed deeper in the kernel, often at syscall handlers or other functions where the arguments need to be scrutinized. Also, a mapping of function argument are needed since some of the syscall arguments are resolved to internal data structures when being passed to the security hooks (e.g., a file descriptor argument is resolved to `struct file *`).

Deploying Sifter filters beyond Android. Sifter is built on top of eBPF, which is originally developed for Linux. Thus, Sifter is not limited to Android and can be used for all devices using the Linux kernel. Moreover, due to the popularity of eBPF and its ability of extending kernel functionality, Microsoft is also adopting it in Windows [13]. Therefore, Sifter may be applicable to a wider range of mobile devices in the future, such as laptops and embedded devices.

Deploying Sifter filters along with other defenses. Sifter is orthogonal to other kernel hardening techniques. Since kernel hardening techniques are developed to prevent specific types of attacks, multiple of them are usually present on a system to make the kernel harder to exploit. Sifter as an attack surface reduction approach may prevent vulnerabilities from being triggered in the first place, so that even if a vulnerability cannot be mitigated by other techniques in the later stage of the exploitation, the kernel will remain safe.

3.10 Related Work

Syscall filtering. There is a long line of work on syscall filtering frameworks [47, 48, 66, 25, 91, 59, 87, 42]. These filtering frameworks use different hooks, which give them different capabilities and performance implications. ptrace-based approaches are generally slower due to their inability to trace syscalls selectively [25]. Mbox [66] utilizes seccomp/BPF to only monitor syscalls of interest and then invoke the ptrace-based tracer to query the policy engine, which greatly improves performance. However, they all require a policy engine running in the user space. As a result, to check a syscall, at least two context switches need to be made. In addition, more ptrace syscalls are needed to inspect the user space memory or register content in the tracee. These context switches make the approach less efficient. Compared to Mbox, Sifter's eBPF approach does not result in context switches since the eBPF filter runs within the same context as the syscall-calling process. User memory probing helper functions also eliminate the need for additional context switches.

SELinux [98] also has the ability to filter syscalls. With extended permission access vector rules, it can also check some arguments of syscalls. For example, the command argument of `ioctl`. However, it cannot probe and check deeper into in-memory arguments. Without the ability to inspect in-memory arguments, 4 out of 5 vulnerabilities mitigated by Sifter's argument limiting policy would not have been mitigated. Also, it does not provide memory to construct stateful policies.

Moreover, to the best of our knowledge, none of existing works targets security-critical kernel modules, which requires low performance overhead from the filtering solution.

Kernel specialization. To reduce the attack surface of the kernel, kernel specialization methods restrict access to the kernel code for target applications by debloating it. KASR [120] first uses offline training stage to generate the kernel code usage database for a specific application in a trusted hypervisor. It then uses runtime enforcement to remove

access permission to unused code of executable and selectively activate the corresponding used code. Face-Change [58] differs from KASR by supporting multiple applications running together. Using a hypervisor, it creates specialized kernel views for each of the target applications and switches between them. MULTIK [68] also supports multiple applications by orchestrating multiple kernels specialized for applications, but avoids the overhead of virtualization and runs natively on the system. Compared to these three schemes, SHARD [16] significantly reduces the attack surface by specializing at both the application and system call levels and strictly enforcing debloating in a more fine-grained way. A limitation of kernel specialization is that it cannot mitigate a vulnerability if the vulnerable code is needed and used by the application. Sifter, however, can mitigate such vulnerabilities.

Policy generation. To generate the syscall filtering policy without domain knowledge, Sifter takes the approach of inferring it from a set of programs, which is similar to systrace [87]. However, to better capture the behavior of programs, systrace translates syscalls with arguments to higher-level information with semantics. This approach requires the developer to manually define the semantics and craft the translation logic. Although Sifter lacks the complete semantics, it does not blindly restrict syscalls arguments to known values. With the syscall definition provided by syzkaller, we define different argument policies for different type of arguments. Moreover, Sifter tries to automatically identify higher-level operations consisting of multiple syscalls.

SPOKE [111] is a policy generation framework for SEAndroid. It extracts the domain knowledge from functional tests since they could carry rich semantics. Access patterns from multiple layers (e.g., Dalvik layer, native layer, and kernel layer) are collected, correlated, and noise-filtered to form a knowledge base. In contrast, Sifter collects syscall usage traces from real programs. Abhaya [85] uses interprocedural static analysis to generate syscall policies automatically. They do not however generate filters for kernel modules.

Offline intrusion detection. A long line of research has been conducted on detecting

intrusion or anomaly [46, 60, 113, 92, 79, 31, 76, 28, 50, 51]. Syscall-based intrusion detection [46, 60, 113, 79, 28, 50] works by first capturing normal program syscall behaviors using data models [46, 60, 113], classification algorithms [79] or machine learning techniques [51]. Then, at the next stage, it utilizes a syscall interposition mechanism to track syscalls of an interesting process and check against the model. Although some of the techniques are shown to be effective, they often run the analysis offline (i.e., syscall traces are analyzed after being made) in the user space due to their complexity. Therefore, it leaves a vulnerability window for an adversary to perform an attack before being caught. Sifter does not have this problem because it directly deploys its policies in the kernel and check syscalls against those policies at runtime. In Sifter, we show that policies we use incur little performance overhead and they stop a large number of CVEs from being exploited without *a priori* knowledge of these vulnerabilities.

Android Malware detection. In Android, app marketplaces such as Google Play, Amazon Appstore, and T-Market perform malware detection to prevent spreading them. Similar to offline intrusion detection, a model is trained using different techniques and scanning is performed when a developer submits an app. APIChecker [55] used by T-Market takes a deny-list approach by training a API-usage-based classification model using known good apps and malicious apps. In Sifter, we take an allow-list approach: we only use a set of good apps and assume zero-days are not yet known. In addition, Sifter is a runtime defense compared to the one-time scan. A malicious app can evade "scan once and allow" approaches if the analysis fails to cover the code containing malicious API calls, or the malicious app is able to detect the sandbox environment and stay dormant during the analysis. In fact, [67] has demonstrated that creating an undetectable sandbox environment is extremely hard. Therefore, our runtime approach that does not require knowledge of vulnerabilities is orthogonal to APIChecker.

Chapter 4

BRF: eBPF Runtime Fuzzer

4.1 Introduction

Extended Berkeley Packet Filter (eBPF) is a rapidly evolving technology in the Linux kernel that enables generic programmability in the kernel space. Originally, the Classic Berkeley Packet Filter (cBPF) was developed specifically for filtering network packets. It allowed user-supplied programs to be loaded and executed in the kernel space to inspect packets and decide whether to allow or reject them. A virtual machine in the kernel interpreted the simple cBPF bytecodes of the program. In 2014, with new instructions and an enhanced virtual machine, eBPF was introduced [101]. As a result of the greater generic programmability and persistent data storage (i.e., maps) [100], it has quickly gained traction in different domains in the kernel. Not only is it widely adopted in different places in the networking stack, it is also integrated with kernel tracing, debugging, and auditing frameworks.

While eBPF brings extensibility and performance to the Linux kernel, it also introduces risks as user-supplied programs run in the kernel. To ensure that an eBPF program, potentially supplied by attackers, can run safely in the kernel space, the eBPF verifier checks the program

before loading it to make sure it will not execute indefinitely or access invalid memory. Obviously, the correctness of the verifier is critical to the safety of kernel. Therefore, many recent works [49, 112, 80, 24, 106, 108, 7, 81, 41, 93] have proposed different ways to test or verify it.

However, we note that eBPF has a significant footprint in the kernel beyond the verifier, i.e., the eBPF runtime, which executes an eBPF program after it passes the verifier. The runtime components mainly consist of the execution environment (i.e., the interpreter or the just-in-time (JIT) compiler) and functionality that cannot be realized using only eBPF bytecodes (e.g., helper functions and maps). Therefore, it is critical to find and fix the vulnerabilities in the eBPF runtime to prevent exploits.

Fuzzing is a promising technique to find such vulnerabilities. Unfortunately, our experiments with the state-of-the-art kernel fuzzer, Syzkaller, shows that it cannot effectively fuzz the eBPF runtime for two reasons. First, the aforementioned eBPF verifier rejects many fuzzing inputs because (1) they do not comply with its required semantics or (2) they miss some dependencies, i.e., other syscalls that need to be issued before the program is loaded. Second, Syzkaller fails to attach and trigger the execution of eBPF programs most of the times. We briefly discuss these issues here.

Program semantics. Similar to other language processors, inputs to the eBPF subsystem (i.e., eBPF programs) pass through both syntax and semantic checks, and are then converted into low-level machine code (if JIT is enabled). More specifically, input eBPF bytecode first go through a series of checks in the verifier. Not only does the bytecode have to conform to the eBPF instruction set format (i.e., syntax), the control and data flow also need to conform to requirements imposed by the verifier to safeguard the kernel (semantics).

Program dependencies. In eBPF, a series of preparatory syscalls are often needed to be made in order to correctly load the bytecode into the kernel. The sequence and arguments

of these syscalls may depend on the eBPF program. For instance, loading a program that uses maps will require making syscalls to create compatible eBPF maps in the kernel in the first place. The program also needs to be rewritten to refer to the external resources such as the just created maps, which is referred as *relocation* in eBPF.

Program execution. Finally, loaded eBPF programs need additional syscalls to be executed. Since eBPF programs are triggered by kernel events, they first need to be correctly attached to the entry points. Then, corresponding events need to created to trigger the execution of the programs.

To demonstrate Syzkaller's ineffectiveness in fuzzing the eBPF runtime, we perform an experiment. During a 40-hour fuzzing session, only 18.1% of `BPF_PROG_LOAD` syscalls succeed in passing the verifier. And the programs that pass the verifier are simple. For example, the average number of instructions in a program is 0.42 (showing that many of the successfully loaded programs are actually empty). Moreover, only 3,648 programs are actually executed after being loaded!

In this work, we set out to tackle these challenges to effectively fuzz the eBPF runtime. Our solution is BRF[1], a fuzzer which is able to generate fuzzing inputs that, on the one hand, satisfy both the eBPF semantic constraints and eBPF program dependencies, and on the other hand, are expressive enough to explore different execution paths within the runtime. Moreover, BRF attempts to attach and execute the successfully loaded programs.

We address three important challenges in BRF. First, BRF produces semantic-correct eBPF programs efficiently by using source-code level input generation/mutation and built-in semantic rules. By generating/mutating eBPF programs at the source-code level and compiling them into actual fuzzing inputs, the compiler will automatically take care of the basic semantics of a program. For example, a branch instruction will not jump to invalid locations,

[1]We will open source BRF.

or an instruction will not access invalid stack memory unless we explicitly perform pointer arithmetics. To further comply with additional semantics imposed by the eBPF verifier, we study the verifier, extract the rules and integrate them with the program generation/mutation logic. Second, to tackle the challenge of syscall dependencies to eBPF programs, we study the relationship between the preparatory syscalls and eBPF programs, and then, in addition to random syscalls, we also generate preparatory syscalls with constrained arguments. Finally, we generate syscalls to attach and trigger the eBPF programs.

Using extensive experiments, we show that 98% of the fuzzing inputs generated by BRF succeed in passing the verifier. Moreover, a large percentage of these programs are successfully attached to corresponding entry points and subsequently executed. Overall, in 40-hour fuzzing sessions, BRF manages to execute 35× more eBPF programs than Syzkaller. Furthermore, the programs successfully loaded by BRF are more expressive compared to programs successfully loaded by Syzkaller, i.e., they include 39× more instructions, 178× more calls to helper functions, and 91× more use of maps. As a result, BRF can cover 52.4% more basic blocks in the eBPF runtime when compared with Syzkaller. Finally, BRF has so far managed to find 4 new vulnerabilities (3 of which are assigned CVE numbers), proving its capability in finding vulnerabilities in the heavily-shielded runtime components.

4.2 Background

4.2.1 Workflow of eBPF

We have witnessed a wide and rapid adoption of eBPF in areas such networking, tracing, and security [14].For different use cases, there exist corresponding program types (e.g., `BPF_PROG_TYPE_SOCKET_FILTER` for filtering packets and `BPF_PROG_TYPE_LIRC_MODE2` for decoding infrared signals). The program types limit the resources they can access (e.g.,

BPF_PROG_TYPE_LIRC_MODE2 should not be able to access network packets). We further breakdown the workflow into three phases: loading, attaching, and execution.

Loading. An eBPF program that a user wants to execute first needs to be loaded into the kernel. This involves loading BPF type format (BTF) information, creating eBPF maps, relocating the eBPF program and finally loading the program. First, since an eBPF program may point to some variables in the kernel, BTF information is needed to resolve the references. Next, eBPF maps used in the eBPF program need to be created by calling BPF syscalls. Then, an eBPF program with instructions referring to external resources (i.e., kernel variables, maps) needs to be rewritten to point to the actual resources, which is also known as relocation. Finally, the eBPF verifier checks the program for the safety. After that, it will be compiled by the JIT compiler if enabled. The kernel will return a file descriptor of the loaded program on success.

Attaching. Once the eBPF program is loaded, the user can attach it using the aforementioned file descriptor to hooks in the kernel, which will be the entry points of the program. The hooks can be functions in the network stacks, kernel functions, security hook, etc.

Execution. Once the aforementioned hooks are triggered in the kernel, the attached program is executed either by an interpreter or natively if the in-kernel eBPF just-in-time (JIT) compiler is enabled. Depending on the hook, different data will be passed to a program as the argument, which is called *context*. During the execution, the eBPF program can store or read data in eBPF maps. It can also interact with the kernel through a predefined set of *helper functions* in the kernel, which allow a program to, for example, access maps, retrieve kernel information, or print messages. Finally, an eBPF program may return an integer value, which will be interpreted by the kernel according to the program type. For example, a socket filter program may return 0 to instruct the kernel to drop the packet.

4.2.2 eBPF Verifier

To ensure programs supplied by user space programs can be safely executed in the kernel, eBPF verifier statically checks them during loading. Specifically, it mainly prevents unbounded execution, invalid jump, invalid memory access, and leak of sensitive kernel data. To do so, the verifier first checks if a program contains loops in the control flow. Then, it walks through the instructions and performs checks specific to the type of the instruction. As it traverses the program, the verifier keeps track of the register state, which includes the potential ranges and types of values in the registers. Some examples of the value types are normal scalar values (`SCALAR_VALUE`), pointers to program context (`PTR_TO_CTX`), pointers to map (`CONST_PTR_TO_MAP`), and pointers to stack memory (`PTR_TO_STACK`). As a result, the verifier is able to determine whether a pointer dereference is safe. For example, an instruction will only access the valid fields in the context with correct permission, or only pointers to valid stack memory can be dereferenced. In addition, since the verifier can track the propagation of pointer values, leaking them to the user space (e.g., directly or indirectly through helper functions and return values) can be prevented. We will discuss the verifier rules in more depth in §4.4.

4.2.3 BPF Runtime

We define the eBPF runtime components as the parts of the eBPF subsystem that are executed once a eBPF program pass the safety checks from the verifier. We identify four major components: JIT compiler, interpreter, eBPF maps, and helper functions.

JIT compiler. In systems running on supported architectures (e.g., x86 and ARM), verified eBPF programs can be further compiled into native machine code to speed up the performance by the JIT compiler built into the kernel.

Interpreter. If JIT is disabled or not supported by the architecture, the eBPF interpreter will be in charge of execution of programs, i.e., decoding eBPF bytecodes on the fly and executing them.

eBPF maps. One significant improvement of eBPF over cBPF is persistent storage that preserves data even after programs terminate. There are 29 different types of eBPF maps provided in Linux v5.15. In general, they are key-value stores but differ in the underlying data structure (e.g., hash table, array, or ring buffer) or the type of elements being stored, e.g., generic data type, socket, or control group (cgroup). In addition to eBPF programs, maps can also be accessed by user space programs using BPF syscalls.

Helper functions. eBPF programs can interact with the kernel through helper functions, which are a predefined set of functions hard-coded in the kernel. There are 175 helper functions in Linux v5.15. A major portion of the helpers are used for accessing or manipulating eBPF maps. For example, `bpf_map_lookup_elem, bpf_map_update_elem,` and `bpf_map_delete_elem` are used for retrieving, modifying, and deleting an element in a map. Some examples for other helper functions include printing debug messages, getting the `task_struct` of the current task, and redirecting a packet to another network device. Note that the availability of helper functions depends on the program type as their usages only make sense under certain scenarios and privileges. Also, unlike eBPF maps, helper functions can only be invoked by eBPF programs.

We design BRF to fuzz all these runtime components. We specifically think that having the ability to fuzz eBPF maps and helper functions are important since as the eBPF technology finds new applications in the kernel, the number of eBPF maps and helper functions will for sure continue to grow.

4.3 Overview

4.3.1 Goals

In this work, we aim to fuzz the eBPF runtime components. The runtime primarily consists of the JIT compiler, the interpreter, the eBPF maps, and helper functions. To achieve this, we have three goals.

Goal I: Generating semantic-correct eBPF programs. Since most of the runtime components in the kernel space are only accessible through eBPF programs (except maps) instead of BPF syscalls from user space programs, we need to generate eBPF programs and let them test these components. Because the eBPF verifier enforces additional semantics in addition to C semantics on the eBPF programs for safety reasons and reject incorrect programs, the randomly generated eBPF programs need to be semantic-correct in order to pass the heavy scrutiny of the verifier.

Goal II: Generating fuzzing inputs that meet syscall dependencies. A fuzzing input generated by a fuzzer is a program consists of syscalls. For the randomly generated eBPF programs, to pass the verifier to be loaded into the kernel, a series of syscalls need to be made. The sequence and arguments of syscalls depend on the eBPF program. Therefore, to effectively test the runtime, a fuzzing input should at least contain these syscalls that satisfy the eBPF program dependencies.

Goal III: Generating syscalls that attach and trigger eBPF programs. After eBPF programs pass the verifier, to execute them, they first need to be attached to the hooks. Then, events corresponding to the hooks need to generated in order to trigger them. Thus, a fuzzing input should also include syscalls that can attach eBPF programs and trigger the execution.

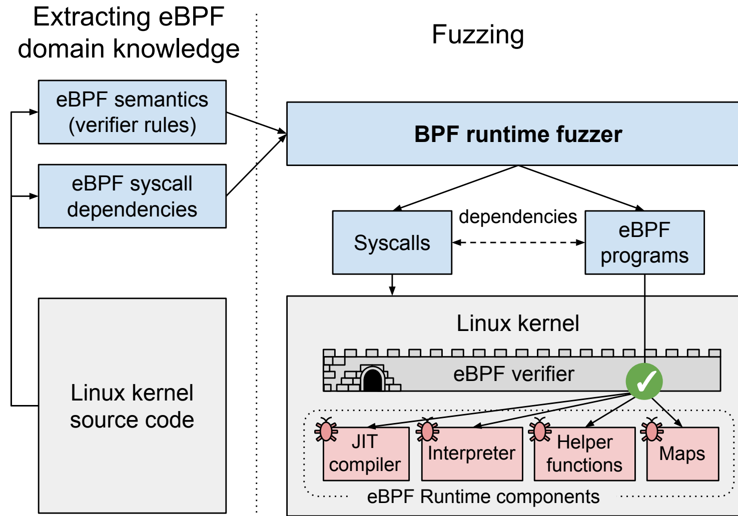


Figure 4.1: The design of BRF

4.3.2 Design

Figure 4.1 shows the high-level idea of BRF. We first extract the eBPF domain knowledge, which includes the eBPF semantics and the syscall dependencies, through manual study of the source code and scripts that automatically parse the source code. Then, during the fuzzing phase, by leveraging the extracted domain knowledge, BRF is able to generate both semantic-correct eBPF programs. It then generates the fuzzing input, which includes not only the eBPF program, but also the required syscalls to correctly load, attach, and execute the eBPF programs. This way, BRF can reach deeply behind the eBPF verifier and effectively fuzz the BPF runtime.

4.3.3 Workflow

BRF is a coverage guided fuzzer as shown in Figure 4.2. In the main fuzzing loop, a fuzzing input, which consists of an eBPF program and a series of required syscalls, is generated. This is done in two steps. First, the scheduler may decide to generate a new eBPF program or mutate an eBPF program in the corpus. Then, different syscalls are generated to form a

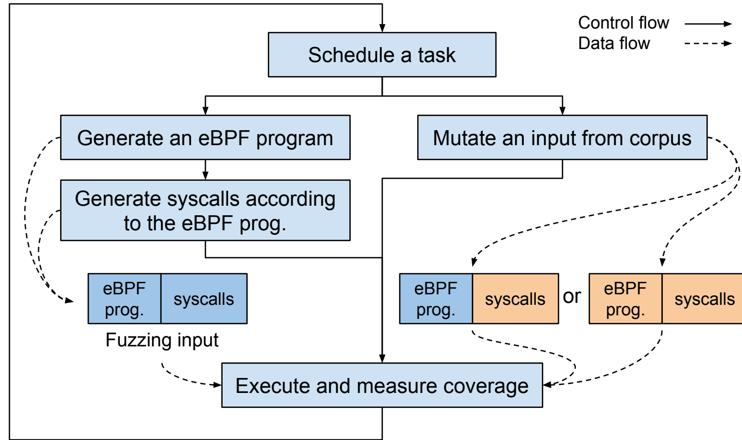


Figure 4.2: The workflow of BRF, where the orange blocks denote the mutated inputs

fuzzing input.

In the first step, to generate a new semantic-correct eBPF program, BRF first randomly chooses a program type and a target helper function since helper functions are one of the key runtime components that we want to test. Then, it tries to generate the arguments of the helper function. For each argument, it will generate the value based on the type. In general, there are three ways to generate a specific type of argument: First, some type of arguments can be generated by directly passing different values into the argument. Examples are scalar values or pointers to stack memory. Second, accessing different fields of the program context sometimes can result in different types of values. For instance, an socket filter eBPF program has `struct sk_buff` as the context. Accessing the field `sk` in this structure will yield a `PTR_TO_SOCK_COMMON` value. Lastly, helper functions can return different types of values. Therefore, BRF may generate other helper function calls and their arguments recursively. Note that the generation logic are constrained based on the rules enforced by the verifier (§4.4) so that the resulting eBPF program will not contain incorrect semantics. The generated eBPF program source code will be compiled into eBPF bytecode and also serialized and stored into the corpus.

In the second step, BRF generates a user space program with different syscalls (i.e., fuzzing

Type of error msg.	Syntax	Semantic	Verifier error	Other
# error msg.	34	226	38	20

Table 4.1: *Types of error messages in the eBPF verifier.*

input). First, syscalls necessary to load, attach and execute the eBPF program are generated. The arguments of these syscalls are generated according to the eBPF program so that they are compatible (§4.6). Then, other random BPF syscalls are generated and appended to the fuzzing input as they can also access some runtime components.

After the eBPF program and the syscalls, serving as a fuzzing input, are generated, it will be executed. During the execution, coverage information is gathered to guide the fuzzing. That is, if a fuzzing input triggers new coverage, the input will be added to the corpus and mutated to facilitate exploring different execution paths.

If the scheduler chooses to mutate a fuzzing input from the corpus, it may decide to mutate the eBPF program or the syscalls. To mutate an eBPF program, BRF first randomly selects an argument of a helper function call. Then, the argument will be randomly generated again using the same logic for input generation. The mutated source code will then be compiled to form the new mutated eBPF program. Since maps may be inserted or removed from the mutated eBPF program during this process, the syscalls associated with the eBPF program also needs to be modified accordingly so that it can still load the eBPF program correctly. On the other hand, if the scheduler decides to only mutate the syscalls, only the randomly generated syscalls and their arguments can be mutated to ensure the eBPF program can still be loaded after the mutation.

4.4 Generating Semantic-Correct eBPF Programs

To improve the effectiveness of the fuzzer in testing runtime components, generating semantic-correct eBPF programs is critical. Although eBPF programs can be written in C and com-

piled into executable using LLVM, the verifier imposes many additional semantics to ensure they can be executed in the kernel space safely. Not only are there many semantic checks, they are mostly *path sensitive*, making the verifier logic very complex. In the kernel source code, the main file that contains verifier logic exceeds 15,000 line of code, not to mention many of other program-specific or map-specific logic scattered elsewhere in the kernel.

Key approach. Studying the verifier in order to identify all the semantic checks is a daunting tasks. Fortunately, we have an observation that allowed us to systematically tackle this challenge. More specifically, we noticed that whenever errors occur in the verifier, corresponding error messages will be printed. As shown in Table 4.1, the error messages mainly include instructions with incorrect syntax, semantic violations, and verifier internal errors. Therefore, our key approach is to use the error messages to enumerate all the semantic rules in the verifier.

Since some verifier rules can be easily satisfied even with random program generation, not all rules need to be taken into consideration. We aim to only find the rules that prevent us from effectively fuzzing the runtime and then constrain our eBPF program generation logic according to these rules. We first annotate the verifier error messages with line numbers and start running the fuzzer without implementing constraints. That is, the fuzzer will randomly generate helper functions and their arguments. Then, during the development of the fuzzer, we add a constraint associated with a verifier rule if the rules is triggered more than once per hour by the eBPF programs generated by the fuzzer. Note that, syntax rules will be automatically satisfied by compiling the source code into eBPF bytecode.

By studying the 226 semantic rules, we are able to constrain the generation and mutation processes of the fuzzer. We explicitly incorporate constrains for 82 semantic rules into the fuzzer; most of the rest will be implicitly taken care of by the compiler and the construct of the program we use. We next discuss some important rules that we have dealt with.

```
1  static const struct bpf_func_proto *
2  kprobe_prog_func_proto(enum bpf_func_id func_id, ...)
3  {
4     switch (func_id) {
5     case BPF_FUNC_perf_event_output:
6        return &bpf_perf_event_output_proto;
7     case BPF_FUNC_get_stackid:
8        return &bpf_get_stackid_proto;
9     ...
10    default:
11       return bpf_tracing_func_proto(func_id, prog);
12    }
13 }
```

Figure 4.3: *Simplified code showing how the verifier checks if a helper is available to kprobe eBPF programs*

4.4.1 Helper Function Availability

For different types of eBPF programs, the helper functions available to them are different. This makes sense as different program types has different use cases and require different privileges. For example, for a socket filer eBPF program that requires almost no special privilege to be loaded, it should not be able to read arbitrary kernel memory using `bpf_probe_read_kernel`. Therefore, when the verifier encounters an eBPF call instruction to a helper function, it calls `get_func_proto`, a member function of the program type, to retrieve the function pointer to the helper function. An example is shown in Figure 4.3, which is the `get_func_proto` of kprobe eBPF programs. If the helper function is not available, the program will be rejected.

To deal with this, BRF first parses the source code of the Linux kernel and looks for the definition of `get_func_proto` for every program type. Sometimes, the `get_func_proto` of a program type may recursively call the `get_func_proto` of another program type. For example, `bpf_tracing_func_proto` shown in Figure 4.3 is the `get_func_proto` of the tracing type eBPF programs. Therefore, after BRF gathers all `get_func_proto` of every program type, those containing other `get_func_protos` will need to be recursively substituted to produce the complete helper availability information. Then, during fuzzing, when BRF

```
1 static const struct bpf_reg_types
2   scalar_types = { .types = { SCALAR_VALUE } };
3 ...
4 static const struct bpf_reg_types *
5   compatible_reg_types[__BPF_ARG_TYPE_MAX] = {
6     [ARG_CONST_SIZE] = &scalar_types,
7     ...
8 }
```

Figure 4.4: *Simplified code showing how the verifier checks if a helper is available to kprobe eBPF programs*

wants to insert a helper function, it will only choose randomly from the ones available to the program types.

4.4.2 Helper Function Arguments

eBPF verifier poses strict type checking on variables in the program. One of such checks happens when passing them to helper functions. As opposed to the C semantics, where one can pass almost any type of variables to function arguments and implicit type conversion will happen under the hood, variables passed to arguments of helper functions have to be of compatible types. For example, as Figure 4.4 shows, a constant size type argument only accepts scalar type variables. This suggests that a variable of any pointer types can never be passed to the argument because it not only makes no logical sense, but also creates a path to leak kernel pointers to the user space.

To generate eBPF programs conforming to these semantics constraints, we first extract the compatibility information between variable types and argument types stored in the `compatible_reg_types`. Besides, BRF automatically parses the kernel source code to extract helper function prototypes, which include the type of arguments. Then, when generating an argument for a helper function, it randomly chooses a compatible type of variable according to the argument type declared in the prototype by directly creating one, calling another helper function, or accessing the program context.

4.4.3 Variable Safety Checks

To prevent unsafe memory accesses commonly seen in programs written in the C language, such as null pointer dereference or out-of-bound access, pointer type variables in eBPF needs to be checked before being dereferenced. By comparing a pointer variable to be a known value or range, the verifier (which is tracking the possible values of variables storing in the register state) is able to determine whether a pointer dereference is safe. Any unsafe pointer dereference will cause the verifier to reject the eBPF program. The safety constraint not only applies to normal expressions but also to arguments of helper functions as pointers can be passed to them and then dereferenced in the kernel.

In detail, there are three different types of checks that are required to safely use different types of variables when passing them as helper function arguments.

Pointer. In the eBPF verifier, there are types of pointers that point to external resources, such as map values, sockets, memory locations or buffers. To make sure null pointer dereference will not happen in helper functions, these pointers need to be compared with `nullptr` before passing them as arguments.

Size. When a helper function takes a pointer to an external memory region as an argument, the argument following the pointer will be the size which will be used in the helper function to access the memory. The size needs to be compared with a constant value that is smaller than the valid range of the memory to make sure out-of-bound memory access will not happen. Also, it might need to be compared with zero when the argument does not allow a zero size access.

Packet. For a pointer to a external packet, it needs to be compared with pointers to the start and end of the packet to make sure the access is limited within the packet.

To satisfy these constraints, BRF keeps track of the valid values of pointers. Then, the safety

checks will be generated before using them as arguments in helper functions.

More specifically, when BRF tries to generate a helper function, it will first generate it arguments, which could come from the return value of another helper function, the program context or direct allocation. Then, an `if` condition block wrapping the helper function call will be generated. The predicates will be filled with the safety checks of the variables passing to the arguments `and`ed together, so that only when all arguments are safe to use, the helper function can be invoked.

For example, when a pointer to a map value is returned from a helper function, BRF knows that the valid size of the pointer should be the size of the value of the map. Therefore, when passing the pointer to another helper function that takes an pointer to a memory region argument and a size argument specifying the size of the memory that will be accessed in the helper function, two checks will be added. First, BRF will add a check to make sure the pointer to the map value return from another helepr is not a `nullptr`. Second, the size argument will be compared to the size of the map value, so that if another random value is passed to the size argument and is larger than the size of map value, the function will not be invoked.

Note that to prevent over-constraining (i.e., putting unnecessary checks on arguments), not all pointer arguments needs to be checked. When tracking pointer values, BRF records whether a pointer can potentially be `nullptr` as some helper functions will always return non-null pointers according to the return value annotated in the prototype. Then, BRF generates a `nullptr` check only if an pointer argument does not allow `nullptr` and the pointer can potentially be a `nullptr`.

```
...
v4 = bpf_ringbuf_reserve(&map_1, v2, v3);
if (v4) {
  v5 = bpf_ringbuf_query(&map_1, v4);
}
...
if (v4) {
  bpf_ringbuf_submit(v4, 0);
}
return 1;
}
```

Figure 4.5: *An example of a semantic-correct eBPF program rejected by the verifier.*

4.4.4 Program Context Access

eBPF programs are invoked with program contexts as arguments. Most of these are pointers to different structures depending of the type of the program and where they are attached. Not all members within the structure may be read or written, and the accesses to the contexts will be checked by the verifier by calling `is_valid_access` to make sure they are not only within the structure, but also with the right permission.

We extract the permission and definition of contexts in different `is_valid_access`s, and use this information to generate random but valid context accesses. When generating a variable for an argument by accessing contexts, BRF first determines if the argument is going to be read or written. This depends on if the argument is of `ARG_PTR_TO_UNINIT_MAP_VALUE` type, which suggests the pointer will be accessed in raw mode and could be written. Then, only the fields with the correct permission are used for random selection.

Note that unlike other variables that are declared and assigned right before the helper, there should be only one variable that accesses a specific field. Otherwise, the compiler optimization will introduce pointer arithmetic on the pointer to context, which violates another verifier rule. Therefore, variables generated by BRF using context accesses are inserted at the beginning of the program and then reused when needed.

4.4.5 Reference

eBPF programs have the ability to acquire references to some kernel resources through helper functions (e.g., acquiring a reference to a socket interface). Therefore, it is important that when a program terminates, the reference to the resources are relinquished.

In BRF, in order to generate eBPF programs that satisfy the reference rules, a fix-up process is performed after an eBPF program is generated. More specifically, BRF goes through the helper functions in a program. If a reference acquired by a helper function never gets released by another helper function, a new reference-releasing helper function will be generated and inserted to the program. On the other hand, when a reference-releasing helper function tries to release a variable that is not produced by a reference-acquiring helper function, BRF will add a helper function that returns a reference and then substitutes the original argument.

When developing BRF for this rule, we notice some false positives of the verifier. That is, an eBPF program conforming to the safety requirement is rejected due to the limitation of the verifier implementation. An example is shown in Figure 4.5. In line 2, `bpf_ringbuf_reserve` will acquire a reference to an entry in a ring buffer, and therefore must be released before the program exit. Although `bpf_ringbuf_submit` is called to release the reference in line 8 after checking if the reservation succeeds in line 7 (which is a necessary safety check), the verifier will determine that there is reference leak if `v4` is null. Therefore, instead of releasing the reference before the program exit, we generate the reference releasing function right after the use, which in this case is after line 4.

4.5 eBPF program dependencies

Being semantic-correct alone does not guarantee that an eBPF program will be successfully accepted by the verifier. It is also necessary for the program to have the correct preparatory

syscalls to be called beforehand in order to load it into the kernel. These syscalls depend on the eBPF program to be loaded, which include BPF syscalls and other generic syscalls. More specifically, in the loading process, these syscalls need to create compatible eBPF maps and relocate the eBPF program. Here we describe these syscalls and their dependencies to the eBPF program.

4.5.1 Creating Compatible Maps

Among the preparatory syscalls, `BPF_MAP_CREATE` syscalls need to be first called to create eBPF maps referred by eBPF programs. The arguments of the syscall decide the attributes (i.e., the type of the map, type of the key, type of the value, the maximum number of entries and the flag describing other properties) of the map to be created. Creating a compatible map can be done in two steps: (1) selecting a compatible type of map and (2) generating valid attributes for the map. The compatibility of the type of map depends on the type of the program and the helper function that takes the map as the argument. The attributes of a map are constrained by the type of the map in addition to the program type and helper functions. Failing to meet the constraints can result in the failure of map creation. In some cases even if the map creation succeeds, the verifier later will reject the program during load time. We further describe the constraints (i.e., the dependencies between programs and maps) and how BRF satisfies them.

Helper function. During fuzzing input generation, an eBPF map is first generated when a helper function wants to use it as an argument. A helper function may only be compatible with a certain type of maps. An obvious example is that `bpf_ringbuf_output` will only accept `BPF_MAP_TYPE_RINGBUF` type of maps.

Thus, when selecting the type of map to be generated, BRF randomly chooses a compatible one using the compatibility information extracted from the verifier in the function

check_map_func_compatibility.

Map type. For different types of maps, the attributes have different constraints and will be checked during the creation. For example, since a BPF_MAP_TYPE_CGROUP_STORAGE map only holds an entry local to a cgroup and is indexed by a 64-bit cgroup ID or struct bpf_cgroup_storage_key, the key size can only be 64 bits or the size of a struct bpf_cgroup_storage_key. Besides, since the number of entries cannot be determined by user space programs, it should pass 0 as the number of max entries in the arguments.

Therefore, to make sure BPF_MAP_CREATE syscalls succeed, BRF generates the attributes according to the constraints of different map types. There are four attributes; and the constraints we extract from the map_alloc and map_alloc_check functions of different map types can be generally described as followed: For the size of keys and the size of values, the constraints are the minimum size, maximum size and the alignment. For the flags, valid flags for a specific map are first separated into groups, where only one flag can only be selected in each group. Then, these selected flags are or-ed together to form the final flag value. Finally, the constraint of the max entries is a single value that limits the upper bound of the random generated value.

Program type. The type of a program may also affect the types of maps that can be used (i.e., certain maps are not compatible with certain program types). For example, a tracing type program that will be attached to hooks that may sleep during execution can only use some basic hash and array maps. The type of program may also affect the flags of the map. For instance, the memory should be pre-allocated for perf event type eBPF programs, BPF_PROG_TYPE_PERF_EVENT. Therefore, the flag, BPF_F_NO_PREALLOC, shall not be used.

When generating a map, BRF first only selects from types of maps that can be used under the current program type, which is similar to how the compatibility of maps with helper functions is handled. Then, attributes are generated based on the constraints coming from

the map type. For the attribute constraints introduced by the program types, BRF tries to fix the flags after all the attributes are generated depending on the programs type.

4.5.2 Relocating eBPF programs

eBPF programs need to be relocated when they call other eBPF programs or there are instructions that refer to maps, external symbols, or global variables. In such case, a user space program that wants to load the program first needs to create the maps or programs or resolve the reference. Then, the references in the instructions need to be updated.

BRF addresses the problem by inserting correct and immutable syscalls into the fuzzing inputs. To generate the correct syscalls, we leverage an eBPF utility library, `libbpf`. We directly invoke the API of `libbpf` that makes syscalls to load an eBPF program after parsing eBPF bytecode and other sections in the ELF file of the program. Note that these preparatory syscalls in the fuzzing inputs will not be mutated. As a result, instead of randomly generating these syscalls and hoping they are sufficient for loading an eBPF program, we can make sure an eBPF program will be loaded successfully first in each fuzzing input.

4.6 Executing eBPF programs

To execute the eBPF programs generated by BRF, two steps need to be done by the fuzzing inputs. eBPF programs supplied by user space programs will not be automatically executed after being successfully loaded into the kernel. They first need to be attached to hooks in the kernel. Then, when the events corresponding to the hooks happen, they will be executed.

4.6.1 Program attachment

For some eBPF program types, the entry point information (i.e., where the program should hook to) is contained within the binary and therefore can be directly attached by calling `BPF_PROG_ATTACH` without specifying them in the arguments. A customized section in the ELF binary of a program is used to specify the type of the program. Sometimes, it may also contain the entry points. For instance, an eBPF program with an ELF section named `kprobe/sys_nanosleep` not only tells the user space programs loading it that it is a kprobe/eBPF program, it also says that the program should be attached to the entry point of the syscall, `nanosleep`. In this case, BRF will use a fixed hook for every program types when generating the binary.

However, for some types of the programs, for flexibility reasons, the entry point information are not be specified in the binaries. Instead, they are provided during attachment using the arguments of the syscall. An example is `BPF_PROG_TYPE_LIRC_MODE2`, which enables decoding infrared signal using eBPF programs. When attaching an LIRC eBPF program, an opened instance of the LIRC driver should be passed to the attach syscall to indicate from which devices the signal should be decoded by the program. Therefore, to properly attach these types of eBPF programs, BRF further generates syscalls required to create and open the resources, and then use it for the attachment.

4.6.2 Triggering the hooks

After the eBPF programs are attached, besides letting them get triggered opportunistically by random events in the kernel, BRF further use two methods to increase the probability of their execution.

First, BRF generates syscalls to trigger 7 types of eBPF programs (4 tracing-related, 2

network-related, and LIRC). For tracing type eBPF programs, we explicitly attach to events that happen frequently (e.g., tracepoints in the kernel scheduler, syscall entry points, or events signaled by hardware periodically). For the two network-related programs, we call `recv` on the socket we created and write data to the socket as well to trigger the execution. For LIRC programs, after attaching to the LIRC device, BRF generates syscalls that write signals to the device, so that the program will be invoked to decode the infrared signal.

Secondly, BRF leverages the BPF syscall `BPF_PROG_TEST_RUN` to deal with eBPF programs associated with subsystems that are hard to set up and test due to their complexity. Introduced for this exact same purpose, `BPF_PROG_TSET_RUN` facilitates testing of a subset of program types by building a simulated environment in the kernel and then test-running the eBPF programs. Therefore, BRF always generates a `BPF_PROG_TEST_RUN` syscall after loading and attaching programs.

4.7 Implementation

We implement BRF on top of Syzkaller in Go and C++ with 3000 LoC. We extracted the constraints from Linux v5.15 by manually studying the verifier and partially parsing the source code automatically. Finally, to collect coverage of eBPF programs for guiding the fuzzing, we extend the coverage collecting framework in the Linux kernel, `kcov`, and instrument the kernel.

4.7.1 Collecting Coverage of eBPF Programs

While Syzkaller already collects coverage information of the fuzzer process to guide the fuzzing, it does not include the coverage of eBPF programs. If not taken care of, eBPF programs that actually trigger new execution paths will not be prioritized, leading to sub-

optimal fuzzing.

To collect coverage information of syscalls issued from the fuzzer process, Syzkaller leverages `kcov` in the Linux kernel. Through compiler instrumentation, a `kcov` function will be invoked for every basic block and edge in the kernel. For every thread that opens and enables `kcov`, the coverage information is then stored to a thread-specific memory region. In other words, only coverage of syscalls from userspace processes are collected.

However, eBPF programs are triggered by different events depending on the program types and the hooks. Therefore, they normally execute in contexts different from the original fuzzer process that loads and attaches the eBPF programs, for example, kernel threads or interrupt contexts. As a result, even when a new helper function is invoked in an eBPF program, the eBPF program may not be prioritized since no positive feedback is generated.

Fortunately, Syzkaller supports collecting extra coverage information related to the fuzzer from other processes by utilizing the *remote* coverage feature of `kcov`. This would allow us to collect a kernel thread's coverage and then associate it with a user process. It requires manually instrumenting the source code, which involves three steps. First, the handle of the user space thread needs to be installed to the kernel thread of interest. We do so when a user space program loads an eBPF program into the kernel space. The handle will be stored into `struct bpf_prog`. Second, in the kernel thread, we need to annotate when to start recording the coverage by calling `kcov_remote_start` with a handle. This is done at the beginning of the function `bpf_prog_run`, with the handle we store in the `struct bpf_prog`. Therefore, the coverage can be associated with the user space program regardless of where the eBPF program is executed. Finally, we need to call `kcov_remote_stop` to stop recording coverage when the execution of the thread leaves the area of interest, which is the end of `bpf_prog_run`.

We further extend the `kcov` remote coverage API to make tracing eBPF program coverage

possible. Since the original remote coverage API allocates a large memory area for storing coverage in `kcov_remote_start` using `vmalloc`, it can only be invoked in context where sleep is permitted. However, this is not the case for most of the eBPF program entry points. Therefore, we create a preallocated version remote coverage API, which takes preallocated memory as argument when starting collecting coverage. Then, for every eBPF program, we allocate a memory region during loading.

4.8 Evaluation

	Program loading syscall			Program attaching syscall			Program execution	
	Total	Total success	Success rate	Total	Total success	Success rate	Total success	Success rate*
Syzkaller	80,230	14,482	18.1%	41,985	10,066	24.0%	3,648	36.2%
BRF	211,847	207,543	98.0%	190,597	180,711	94.8%	127,760	70.1%

Table 4.2: *Comparison of programs loaded, attached, and triggered by Syzkaller and BRF under 40-hour fuzzing sessions. *It is derived by dividing the total number of programs executed by total programs attached.*

4.8.1 Fuzzing Effectiveness

To evaluate the effectiveness of BRF in fuzzing the runtime components of the eBPF subsystem, we compare it with the state-of-art syscall fuzzer, Syzkaller. In the evaluation, we assign the same amount of resources to the two fuzzers, which are five virtual machines, each with 4GB of RAM and eight fuzzer processes. Besides, since we are only interested in fuzzing the eBPF subsystem, only BPF related syscalls are enabled. To have a fair comparison, we enable all BPF-related syscalls for Syzkaller so that it is able to create the necessary resources

	# Instructions		# Helper functions		# Maps	
	Avg.	Max	Avg.	Max	Avg.	Max
Syzkaller	0.42	11	0.03	4	0.03	5
BRF	16.44	260	5.34	59	2.73	18

Table 4.3: *Expressiveness of eBPF program generated and successfully loaded by Syzkaller and BRF.*

for attaching programs and triggering the execution. For example, `socket` and `setsockopt` are enabled so that if a `BPF_PROG_TYPE_SOCKET_FILTER` eBPF program is generated, there are syscalls to create a socket and attach the program to the socket. For BRF, we only enable core eBPF syscalls as the fuzzer will generate the necessary syscalls. In addition, since the JIT compiler is enabled by default on x86 for performance reasons, we report the fuzzing statistics under this configuration. However, when running fuzzing experiments to find vulnerabilities, we try both kernels with and without JIT compiler so that we can also test the interpreter. We run the fuzzer on a machine with Intel Xeon E5-2697 v4 CPU for 40 hours and compare the results.

Reaching the runtime components. First, we look at the three critical stages in the workflow of eBPF that will affect the fuzzing effectiveness: eBPF program loading, attaching and execution. Since many of the runtime components, such as the interpreter, JIT compiler, and helper functions, can only be accessed by eBPF programs, to achieve high fuzzing effectiveness, it is essential first for the programs to pass the verifier during the load time. As shown in Table 4.2, 98% of eBPF programs generated by BRF are able to pass the checks of the verifier, showing that these fuzzing inputs are both semantic-correct and meet syscall dependencies. Since we do not cover all semantic rules, there are still a small portion of fuzzing inputs that fail to pass the verifier. On the other hand, with only syntax-awareness and little knowledge of syscall dependencies, only 18.1% of programs generated by Syzkaller pass the verifier.

Here we report five most violated verifier rules (>100 violations in 40 hour) that bottleneck the fuzzing of the runtime for Syzkaller: 1) calling into invalid destination 2) not having `jmp` or `exit` as the last instruction 3) calling into a `btf_id` which is not a kernel function 4) jumping out of range 5) calling a kernel function from non-GPL eBPF program. Note that these bottlenecks are relatively simple. The reason is that the semantic checks are layered and therefore the complex one are still masked by the simple ones.

After eBPF programs pass the verifier, they will be compiled by the JIT compiler if enabled, and the JIT compiler will be fuzzed. Then, to exercise the interpreter (when JIT is disabled), helper functions and maps, programs first need to be attached. For Syzkaller, due to the fact that only a few eBPF programs are loaded successfully, the program attaching syscall has few valid program `fds` to start with. Since Syzkaller only generates program attaching syscalls by chance, only 24% of these syscalls succeed. In contrast, since BRF generates program attaching syscalls for every fuzzing input, 94.8% of its attaching syscalls succeed. Finally, BRF manages to trigger 70.1% of the attached programs, whereas Syzkaller only manages to trigger 36.2% of the attached programs.

All and all, the advantages of BRF in loading, attaching, and triggering eBPF programs result in BRF executing 35× more programs than Syzkaller.

Expressiveness of eBPF programs. Beside the success rate of loading, attaching, and executing eBPF programs, the expressiveness of the eBPF programs can also affect the fuzzing effectiveness. Therefore, we quantify the expressiveness of the generated eBPF programs by measuring the average number of eBPF instructions, average number of calls to helper functions, and the average number of usage of maps within within successfully loaded eBPF programs. Our results, shown in Table 4.3, show that, on average, a program successfully loaded by BRF contains 39× more instructions, 178× more calls to helper functions, and 91× more use of maps, compared to a program successfully loaded by Syzkaller. The largest program generated by BRF can contain 59 helper functions and 18 maps. The higher effectiveness of BRF in both generating semantic-correct and expressive eBPF programs help it reach broader and deeper in the eBPF runtime.

eBPF runtime coverage. We measure the ability of BRF and Syzkaller in covering different types of eBPF programs, helper functions, and maps. BRF generates 27 types of programs (compared to 20 by Syzkaller); it uses 155 different helper functions (compared to 107 by Syzkaller); and it uses 29 types of maps (compared to 28 by Syzkaller).

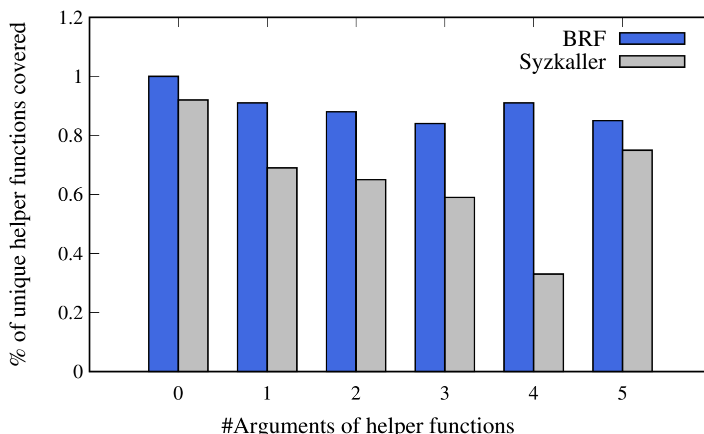


Figure 4.6: *The success rate of BRF and Syzkaller in covering helper functions with a given number of arguments.*

We further look into what types of helper functions are in the programs that pass the verifier. More specifically, we look into the number of arguments of the helper functions. Figure 4.6 shows the percentage of helper functions with a given number of arguments (maximum is 5) covered by BRF and Syzkaller. It shows that as the number of arguments of helper functions increase, Syzkaller does a worse job of covering them. This is because the verifier imposes semantics checks on arguments; thus, as the number of arguments increases, Syzkaller becomes less effective in satisfying the verifier. On the other hand, since BRF is aware of the semantics, the increasing number of helper function arguments affects it much less.

Code coverage. To see how the aforementioned key metrics in the fuzzing effectiveness translate into the ability of exploring more execution paths, we record the coverage of Syzkaller and BRF in the eBPF runtime components. We do so by only including the basic blocks in files that implement the eBPF runtime: JIT compiler, interpreter, helper functions and maps. Therefore, the BPF syscalls, verifier and glue code that connects eBPF subsystem with other subsystems are not included. The result is shown in Figure 4.7, where we find that during the 40-hour fuzzing sessions, BRF is able to continuously discover new paths. In contrast, the coverage of Syzkaller plateau after 5 hours into the fuzzing. At the end of the

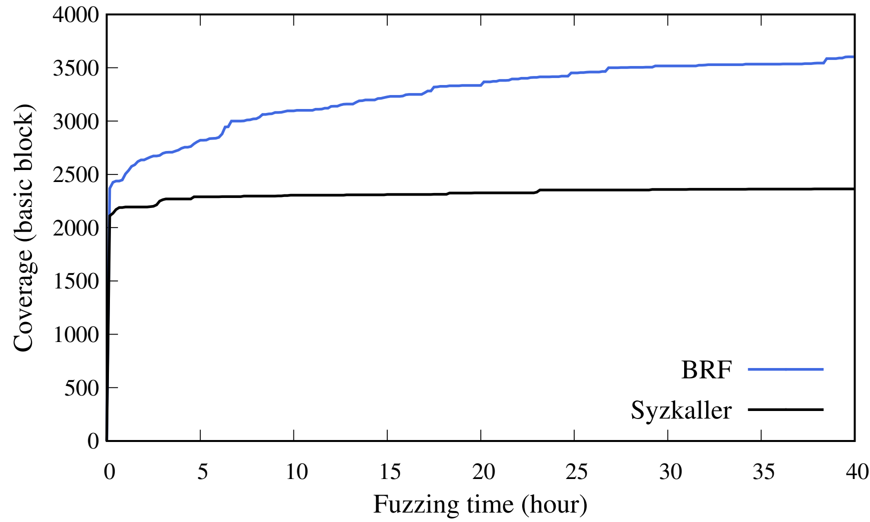


Figure 4.7: *Coverage of eBPF runtime components of BRF and Syzkaller under 40-hour fuzzing sessions*

session, BRF is able to explore 52.4% more basic blocks in the runtime components, showing its better ability in fuzzing eBPF runtime.

4.8.2 Discovered Vulnerabilities

Table 4.4 lists the vulnerabilities found by BRF. Next, we discuss them in detail.

	Verifier	Runtime components				New	Type of vulnerability
		JIT compiler	Interpreter	Helper functions	Maps		
CVE-2022-2905	✓	✓				✓	OOB access, Info leak
CVE-YYYY-XXX2				✓		✓	Deadlock, API misuse
CVE-2023-0160				✓	✓	✓	Deadlock
Vulnerability 4				✓	✓	✓	Deadlock

Table 4.4: *Summary of vulnerabilities discovered by BRF (OOB access: out-of-bound access).*

CVE-2022-2905 is an out-of-bound heap memory access vulnerability. As shown in Figure 4.8, an attacker can probe into the kernel memory by calling `bpf_tail_call` in an eBPF program with a key larger than the `max_entries` of the map. `bpf_tail_call` is a helper function that allows the calling program to jump into another program stored in the map of the kind, `BPF_MAP_TYPE_PROG_ARRAY`. When this malicious eBPF program is loaded into

```
1  DEFINE_BPF_MAP(map_0,
2    BPF_MAP_TYPE_PROG_ARRAY, // type
3    0,                       // map_flags
4    uint32_t,                // key
5    uint32_t,                // value
6    36                       // max_entries
7  );
8  SEC("cgroup/sock_create")
9  int func(struct bpf_sock *ctx) {
10   bpf_tail_call(ctx, &map_0, 49);
11   return 0;
12 }
```

Figure 4.8: *Simplified code illustrating a Proof-of-Concept (PoC) of CVE-2022-2905.*

the kernel, it is able to pass the verifier due incorrect value range analysis in the verifier. Therefore, when the x86 JIT compiler tries to compile the program, it will index the array of program stored in `bpf_array->ptr` using the key. This will cause an out-of-bound access in the heap area as only 36 entries is allocated to `bpf_array->ptr` when creating the map prior to loading the program.

BRF is able to discover the vulnerability because calling `bpf_tail_call` with a key larger than the `max_entries` does not violate the semantic rules enforced by the verifier. Instead, the helper function should perform the check on this argument during run time. Since there is no input generation constraints on the key values, and it is possible for BRF to generate such input. In fact, the verifier would have forced programs to be executed in interpreter mode if the key was not a known constant or was ”out of the range” of the `max_entries` instead of rejecting them. However, since the value range analysis mechanism in the verifier over-approximates the range, a key larger than the `max_entries` could be deemed within the range. As a result, such kind of programs can be executed in JIT mode and trigger the vulnerability.

CVE-YYYY-XXX2[2] is a warning triggered when the helper function, `bpf_probe_read_user`, is called in a non-maskable interrupt (NMI) context. The warning comes from the API for

[2]The CVE number is yet to be finalized as we are still working with the Linux kernel developer in the bug fixing process.

```
1         CPU0                    CPU1
2         ----                    ----
3    lock(&htab->buckets[i].lock);
4                                 local_irq_disable();
5                                 lock(&rq->__lock);
6                                 lock(&htab->buckets[i].lock);
7    <Interrupt>
8      lock(&rq->__lock);
```

Figure 4.9: *lockdep warning showing how CVE-2023-0160 could cause a deadlock.*

copying data from user space memory, `copy_from_user_nofault`, since it was intended to be called in user context. The kernel developer of the eBPF subsystem, Alexei Starovoitov, further found two problems that came along with this mis-used API. First, the user memory access happens without calling `nmi_access_okay`, which could be dangerous on x86 when it is using a different `mm` than the target. Second, the implementation of `copy_from_user_nofault` may try to acquire a spin lock under the configuration `CONFIG_HARDENED_USERCOPY`. If invoked under NMI, this may cause a deadlock.

Triggering the vulnerability requires actually executing the eBPF program under a specific context. Since Syzkaller is not efficient in generating eBPF programs with this helper and triggering them (only 6 programs with this helper are executed by Syzkaller in 40 hours as opposed to 5488 by BRF), it cannot find the vulnerability.

CVE-2023-0160 is a locking problem in the `BPF_MAP_TYPE_SOCKHASH` map and helper functions for manipulating the map. `BPF_MAP_TYPE_SOCKHASH` is a type of hash map provided by eBPF for storing reference to `struct sock`. It protects a bucket in a hash map from concurrent access using locks. The map may be modified using both syscalls from user space programs and helper functions in eBPF programs in contexts that may be interrupted (i.e., the lock is IRQ-unsafe). When an eBPF program that uses this type of map is invoked by an interrupt that already holds a lock, a deadlock could occur.

Figure 4.9 shows a potential deadlock scenario illustrated by `lockdep`, the runtime locking correctness validator in the Linux kernel. First, an interruptible task on CPU0, t_1, acquires

the lock to a bucket, l_1, when trying to modify the map. Meanwhile, an interrupt on CPU1, t_2, that holds another lock, l_2, may be waiting to acquire l_1. Then, if t_1 is interrupted by an interrupt same as t_2 that also tries to acquire l_2, a deadlock will occur since t_1 is interrupted and can no longer release l_1, and t_2 will not be able to release l_2 as it is still waiting for l_1.

Vulnerability 4 is another locking rule violation in the `BPF_MAP_TYPE_QUEUE` map and its corresponding helper functions similar to CVE-2023-0160 that could lead to deadlock. The map provides FIFO storage for eBPF programs and the implementation also uses a lock to protect it from concurrent access. Therefore, in a scenario similar to CVE-2023-0160, a deadlock will occur. lockdep also catches another way it can violate the locking rule. Since the lock can be acquired in interrupt context or in context with interrupt enabled, a task holding the lock can be interrupted by a interrupt that tries to acquire the same lock, resulting in a deadlock.

4.9 Discussions on Future Maintenance of BRF

In BRF, we build the semantic-correct program generation/mutation logic by studying the verifier as well as automatically extracting information (e.g., helper function definitions) needed by the logic. Therefore, as the eBPF subsystem evolves, a question we ask ourselves is how BRF will adapt to changes in the eBPF subsystem, and more specifically, whether the process will require hefty manual efforts. Fortunately, studying the verifier should be mostly a one-time effort given the safety guarantee provided by the verifier should remain mostly the same. As the eBPF subsystem evolves, however, we can expect more program types, helper functions and maps will be added. At that time, only the definition of the newly added program type, helper or map need to be added to BRF in order to support fuzzing them, which can be achieved automatically.

4.10 Related Work

Fuzzing the eBPF subsystem. There are a couple of solutions for fuzzing the eBPF subsystem [7, 41, 81, 93]. BPF fuzzer [7] aims to test the eBPF verifier by leveraging LLVM fuzzer and sanitizer available in the user space. To do so, it compiles the verifier in the user space and let LLVM fuzzer perform mutation-based coverage-guided fuzzing. That is, a set of initial inputs need to be fed to the fuzzer, and new inputs will generated by mutating the initial inputs or corpus. Sample eBPF programs from the Linux kernel source tree are used. Overall, it manages to find one bug.

[81] is a syntax-aware fuzzer targeting the eBPF subsystem in the Linux kernel based on Angora [32], a mutation-based fuzzer that requires a set of initial inputs. Similar to [7], it also uses the sample eBPF programs in the Linux kernel source tree. It tracks interesting bytes that trigger new execution paths and then use gradient descent to guide the mutation. However, it only found bugs in the `libbpf` in the user space.

[93] is a bytecode-level semantic-aware fuzzer targeting the JIT compiler. It works by generating bytecode with some awareness about the semantics imposed on the register states. The generated eBPF program will first be checked by the verifier compiled in the user space using the approach in [7]. If the program is deemed valid by the verifier, it will be loaded into the kernel and JITed to see if it can trigger bugs. The experiments shows only 0.77% of generated eBPF programs are valid due to the limited semantics awareness, and the fuzzer manages to find one bug in the JIT compiler.

Another BPF fuzzer by Crump [41] is a syntax-aware differential fuzzer. By feeding the generated eBPF programs to two different execution environment (i.e., native execution of JITed code and executed by the interpreter) and comparing the results, BPF fuzzer can detect bugs in the runtime if the outputs are different. By using BPF fuzzer on RBPF, a user space BPF runtime implemented in Rust, it found two vulnerabilities.

Compare to previous work, BRF is a semantic-aware and dependency-aware generator-based fuzzer that is able to generate eBPF programs that pass the verifier efficiently. Combining with efforts to attach and trigger the eBPF programs, BRF is able to reach deeply into the runtime and discover 4 new vulnerabilities. To the best of our knowledge, BRF is the first work that cover all major eBPF runtime components.

Formal verification of the eBPF subsystem. In addition to fuzzing, formal verification is another approach used to improve the safety of eBPF subsystem by verifying the correctness of components in the eBPF subsystem or implementing the components with proven correctness [49, 112, 80, 24, 108].

Prevail [49] is an effort in implementing an alternative eBPF verifier in the user space with greater precision, which is adopted by eBPF for Windows [13]. JitK [112] implements an interpreter for cBPF with proven correctness. JitSynth [106] develops a tool that synthesizes eBPF bytecode into verified native RISC-V instructions. Jitterbug[80] models the eBPF JIT compiler in Rosette [105] and then uses it to verify the correctness of JIT compiler implementation of different architectures in the Linux kernel. [24] and [108] try to verify the range analysis mechanism (i.e., `tnum`) using formal verification methods. Interestingly, CVE-2022-2905 discovered by BRF is due to a flaw in `tnum`.

We believe that fuzzing the eBPF runtime is orthogonal to the verification of eBPF. Even in the future, when the verifier, JIT compiler and the interpreter can be implemented with proved correctness, we believe the rest of the runtime components are less likely to be verified due to their diverse functionality and increasing number. Therefore, testing the runtime components should be a complementary effort in making the eBPF subsystem safer.

Fuzzing language processors. Many efforts have been invested in improving fuzzing language processors [117, 61, 107, 32, 18, 33, 83, 86, 99], software that translate source code in high-level languages into lower-level languages. Examples are compilers, JIT runtimes,

and interpreters. As mentioned earlier, eBPF is also a language processor.

These approaches can be categorized into mutation-based and generator-based. For mutation-based fuzzers, initial inputs are required to generate new fuzzing inputs, which often suggests that the fuzzers have limited knowledge about the syntax and semantics. For generator-based fuzzers, with some knowledge about the syntax or semantics, they are able to generate new inputs by themselves.

CSmith [117] is a C compiler fuzzer that performs differential testing. C code is randomly generated according to the grammar and then fed to different compilers. After compilation and execution, the results are compared to determine if there are bugs in the compilers. LangFuzz [61] is a mutation-based blackbox fuzzer that generates and mutates code fragments in the fragment pool, which is constructed by parsing codebases and test suites using a language-specific parser. IFuzzer [107] is a mutation-based fuzzer that try to generate new inputs using genetic programming. Angora [32] aims to improve the branch coverage by introducing several techniques that facilitate constraint solving without using symbolic execution. The techniques are context-sensitive branch coverage, scalable byte-level taint tracking, search based on gradient descent, type and shape inference, and input length exploration. Nautilus [18] is a grammar-based fuzzer that does not rely on corpora. It combines coverage-guided feedback to achieve higher fuzzing performance. PolyGlot [33] improves the generic applicability of a grammar-aware fuzzer by performing mutation and analyses in IR level. It first generates an IR translator using the BNF grammar of the language. Then, inputs selected from corpus are lifted using the IR translator. Constraints mutation will produce syntax-correct inputs and a semantic validator will further fix semantic errors. Zest [83] facilitates fuzzing of the semantic-stage of language processors by preserving the validity of syntax of inputs and using the validity feedback as guidance in addition to code coverage. [86] is a JavaScript fuzzer that introduces aspect-preserving mutation that avoids destroying the semantic of corpus. By stochastically preserving the structures and types in input cor-

pus, it has better chance in generating inputs that are syntax-correct and semantic-correct. Gramatron [99] improves the efficiency of grammar-aware fuzzers by addressing two shortcomings of traditional parse tree-base fuzzer. It uses grammar automaton to avoid biased sampling and aggressive mutation to avoid small-scale mutation.

BRF adopts the generator-based approach with the guidance of code coverage. Besides, due to the complex eBPF semantics imposed by the verifier due to security reasons, we extract semantic rules from the verifier to make BRF grammar-aware, so that the fuzzing inputs are able to reach eBPF runtime efficiently. We believe that these techniques for fuzzing language processors are orthogonal to our work and can be incorporated to BRF in the future to further improve the fuzzing effectiveness.

Chapter 5

Conclusion

Dynamic program analysis is a powerful tool that can help secure operating systems. This dissertation presented three applications of dynamic program analysis in finding bug/vulnerabilities in and hardening the operating system. First, *Mousse* presented a new approach of applying selective symbolic execution on programs with untamed environments (e.g., The HAL programs in the operating system). Using process-level SSE, environment-aware concurrency and distributed execution, *Mousse* achieved better performance and coverage without introducing false positives compared to existing approaches. Second, *Sifter* presented a solution for reducing the attack surface of security-critical kernel modules with low performance overhead and manual effort. Evaluation showed that by learning syscall patterns from legitimate programs, *Sifter* could generate syscall filters that mitigate about half of the vulnerabilities without knowing them in advance. Finally, *BRF* presented a fuzzer that can satisfy the semantics and syscall dependencies required by the eBPF verifier and subsystem in order to test the eBPF runtime components. Evaluation showed that, *BRF* not only can execute 35× more eBPF programs compared to Syzkaller, but the generated programs are also more expressive. Moreover, it covered 52.4% more code in the runtime components. Overall, all the three presented work tackled the challenges of using dynamic program anal-

ysis techniques to secure operating systems.

Bibliography

[1] Android Media Framework. https://source.android.com/devices/media.

[2] The Kernel Concurrency Sanitizer (KCSAN). https://github.com/google/kernel-sanitizers/blob/master/KCSAN.md.

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

[4] Trinity: A Linux System call fuzz tester. https://codemonkey.org.uk/projects/trinity/.

[5] UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.

[6] The Kernel Address Sanitizer (KASAN). https://github.com/google/kasan/wiki, 2018.

[7] Bpf fuzzer. https://github.com/iovisor/bpf-fuzzer, 2019.

[8] Nokia 9 PureView. https://www.nokia.com/phones/en_int/nokia-9-pureview/, 2019.

[9] Symbion: fusing concrete and symbolic execution. https://angr.io/blog/angr_symbion/, 2019.

[10] Mousse source code. https://trusslab.github.io/mousse/, 2020.

[11] Syzkaller: coverage-guided Linux system call fuzzer. https://opensource.google.com/projects/syzkaller, 2021.

[12] BPF Features by Linux Kernel Version. https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md, 2022.

[13] eBPF for Windows. https://github.com/Microsoft/ebpf-for-windows, 2022.

[14] ebpf summit. https://ebpf.io/summit-2022/, 2022.

[15] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert. Intel Virtualization Technology for Directed I/O. *Intel Technology Journal*, 2006.

[16] M. Abubakar, A. Ahmad, P. Fonseca, and D. Xu. SHARD: Fine-Grained Kernel Specialization with Context-Aware Hardening. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[17] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proc. ACM PLDI*, 2014.

[18] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert. Nautilus: Fishing for deep bugs with grammars. In *NDSS*, 2019.

[19] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic Exploit Generation. In *Proc. Network and Distributed Systems Security Symposium (NDSS)*, 2011.

[20] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley. Automatic Exploit Generation. *Communications of the ACM*, 2014.

[21] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX ATC, FREENIX Track*, 2005.

[22] F. Bellard. Tiny Code Generator. `https://git.qemu.org/?p=qemu.git;a=blob_plain;f=tcg/README;hb=HEAD`, 2020.

[23] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B. A. Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proc. USENIX OSDI*, 2010.

[24] S. Bhat and H. Shacham. Formal verification of the linux kernel ebpf verifier range analysis. 2022.

[25] A. Bianchi, Y. Fratantonio, C. Kruegel, and G. Vigna. NJAS: Sandboxing Unmodified Applications in non-rooted Devices Running stock Android. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 27–38, 2015.

[26] C. S. Brown and J. Westenberg. The first phone with an under-glass fingerprint sensor officially announced. `https://www.androidauthority.com/vivo-inscreen-fingerprint-launch-831822/`, 2018.

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

[28] R. Canzanese, S. Mancoridis, and M. Kam. System call-based detection of malicious processes. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 119–124. IEEE, 2015.

[29] L. Ceze, M. D. Hill, and T. F. Wenisch. Arch2030: A Vision of Computer Architecture Research over the Next 15 Years. *A Computing Community Consortium (CCC) white paper*, 2016.

[30] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on Binary Code. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2012.

[31] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):1–58, 2009.

[32] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[33] Y. Chen, R. Zhong, H. Hu, H. Zhang, Y. Yang, D. Wu, and W. Lee. One engine to fuzz'em all: Generic language processor testing with semantic validation. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 642–658. IEEE, 2021.

[34] V. Chipounov and G. Candea. Reverse Engineering of Binary Device Drivers with RevNIC. In *Proc. ACM EuroSys*, 2010.

[35] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea. Selective Symbolic Execution. In *Proc. USENIX Workshop on Hot Topics in System Dependability (HotDep)*, 2009.

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

[37] V. Chipounov, V. Kuznetsov, and G. Candea. The S2E platform: Design, implementation, and applications. *ACM Transactions on Computer Systems (TOCS)*, 2012.

[38] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea. Cloud9: A Software Testing Service. *SIGOPS Operating System Review*, 2010.

[39] J. Corbet. The 5.7 kernel is out. `https://lwn.net/Articles/821829/`, 2020.

[40] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2017.

[41] A. Crump. Earn $200k by fuzzing for a weekend: Part 1. `https://secret.club/2022/05/11/fuzzing-solana.html`, 2022.

[42] DeMarinis, Nicholas, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis. Sysfilter: Automated system call filtering for commodity software. In *In 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[43] S. Dhillon. [PATCH net-next 0/3] eBPF Seccomp filters. `https://lists.linuxfoundation.org/pipermail/containers/2018-February/038476.html`, 2018.

[44] J. Edge. Kernel runtime security instrumentation. `https://lwn.net/Articles/798157/`, 2019.

[45] D. Engler and D. Dunbar. Under-constrained Execution: Making Automatic Code Destruction Easy and Scalable. In *Proc. Int. Symp. on Software Testing and Analysis (ISSTA)*, 2007.

[46] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *1996 IEEE Symposium on Security and Privacy (S&P). IEEE*, pages 120–128, 1996.

[47] T. Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proc. Network and Distributed Systems Security Symposium (NDSS)*, 2003.

[48] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. In *Proc. Network and Distributed Systems Security Symposium (NDSS)*, 2004.

[49] E. Gershuni, N. Amit, A. Gurfinkel, N. Narodytska, J. A. Navas, N. Rinetzky, L. Ryzhyk, and M. Sagiv. Simple and precise static analysis of untrusted linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019.

[50] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis. Temporal system call specialization for attack surface reduction. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1749–1766, 2020.

[51] A. K. Ghosh, A. Schwartzbard, and M. Schatz. Learning program behavior profiles for intrusion detection. In *Workshop on Intrusion Detection and Network Monitoring*, 1999.

[52] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proc. ACM PLDI*, 2005.

[53] P. Godefroid, M. Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *Proc. Network and Distributed Systems Security Symposium (NDSS)*, 2008.

[54] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Communications of the ACM*, 2012.

[55] L. Gong, Z. Li, F. Qian, Z. Zhang, Q. A. Chen, Z. Qian, H. Lin, and Y. Liu. Experiences of landing machine learning onto market-scale mobile malware detection. In *Proc. ACM EuroSys*, 2020.

[56] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, D. Tsafrir, and A. Schuster. ELI: Bare-Metal Performance for I/O Virtualization. In *Proc. ACM ASPLOS*, 2012.

[57] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information Flow Analysis of Android Applications in DroidSafe. In *Proc. Network and Distributed Systems Security Symposium (NDSS)*, 2015.

[58] Z. Gu, B. Saltaformaggio, X. Zhang, and D. Xu. Face-change: Application-driven dynamic kernel view switching in a virtual machine. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 491–502. IEEE, 2014.

[59] M. Harvan and A. Pretschner. State-based usage control enforcement with data flow tracking using system call interposition. In *2009 Third International Conference on Network and System Security*, pages 373–380, 2009.

[60] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion Detection Using Sequences of System Calls. *Journal of computer security*, 6(3):151–180, 1998.

[61] C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458, 2012.

[62] H.-W. Hung, Y. Liu, and A. A. Sani. Sifter: protecting security-critical kernel modules in android through attack surface reduction. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, 2022.

[63] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (S&P). IEEE*, 2019.

[64] D. Jones. Triforce linux syscall fuzzer. `https://github.com/nccgroup/TriforceLinuxSyscallFuzzer`, 2016.

[65] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee. HFL: Hybrid Fuzzing on the Linux Kernel. In *Proc. Network and Distributed Systems Security Symposium (NDSS)*, 2020.

[66] T. Kim and N. Zeldovich. Practical and Effective Sandboxing for Non-root Users. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 139–144, 2013.

[67] B. Kondracki, B. A. Azad, N. Miramirkhani, and N. Nikiforakis. The droid is in the details: Environment-aware evasion of android sandboxes. In *Proc. Network and Distributed Systems Security Symposium (NDSS)*, 2022.

[68] H.-C. Kuo, A. Gunasekaran, Y. Jang, S. Mohan, R. B. Bobba, D. Lie, and J. Walker. Multik: A framework for orchestrating multiple specialized kernels. *arXiv preprint arXiv:1903.06889*, 2019.

[69] A. Kurmus, A. Sorniotti, and R. Kapitza. Attack surface reduction for commodity os kernels: Trimmed garden plants may attract less bugs. In *Proceedings of the Fourth European Workshop on System Security*, EUROSEC '11, New York, NY, USA, 2011. Association for Computing Machinery.

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

[71] J. Liu, W. Huang, B. Abali, and D. K. Panda. High Performance VMM-Bypass I/O in Virtual Machines. In *Proc. USENIX ATC*, 2006.

[72] M. Liu, T. Li, N. Jia, A. Currid, and V. Troy. Understanding the Virtualization "Tax" of Scale-out Pass-Through GPUs in GaaS Clouds: An Empirical Study. In *Proc. IEEE HPCA*, 2015.

[73] Y. Liu, H.-W. Hung, and A. A. Sani. Mousse: A System for Selective Symbolic Execution of Programs with Untamed Environments. In *Proc. ACM EuroSys*, 2020.

[74] Y. Liu, H.-W. Hung, and A. A. Sani. Mousse: a system for selective symbolic execution of programs with untamed environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.

[75] L. Luo, Q. Zeng, C. Cao, K. Chen, J. Liu, L. Liu, N. Gao, M. Yang, X. Xing, and P. Liu. System Service Call-Oriented Symbolic Execution of Android Framework with Applications to Vulnerability Discovery and Exploit Generation. In *Proc. ACM MobiSys*, 2017.

[76] F. Maggi, M. Matteucci, and S. Zanero. Detecting intrusions through system call sequence and argument analysis. *IEEE Transactions on Dependable and Secure Computing*, 2010.

[77] I. Malchev. Here comes Treble: A modular base for Android. `https://android-developers.googleblog.com/2017/05/here-comes-treble-modular-base-for.html`, 2017.

[78] M. Muench, D. Nisi, A. Francillon, and D. Balzarotti. Avatar^2: A Multi-target Orchestration Platform. In *Proc. Workshop on Binary Analysis Research (BAR)*, 2018.

[79] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel. Anomalous system call detection. *ACM Transactions on Information and System Security (TISSEC)*, 9(1):61–93, 2006.

[80] L. Nelson, J. Van Geffen, E. Torlak, and X. Wang. Specification and verification in the field: Applying formal methods to bpf just-in-time compilers in the linux kernel. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, 2020.

[81] B. C. Nilsen. Fuzzing the berkeley packet filter, 2020.

[82] M. Owen. A deep dive into HomePod's adaptive audio, beamforming and why it needs an A8 processor. `https://appleinsider.com/articles/18/01/27/a-deep-dive-into-homepods-adaptive-audio-beamforming-and-why-it-needs-an-a8-proces` 2018.

[83] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 329–340, 2019.

[84] S. Pailoor, A. Aday, and S. Jana. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 729–743, 2018.

[85] S. Pailoor, X. Wang, H. Shacham, and I. Dillig. Automated policy synthesis for system call sandboxing. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–26, 2020.

[86] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim. Fuzzing javascript engines with aspect-preserving mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1629–1642. IEEE, 2020.

[87] N. Provos. Improving Host Security with System Call Policies. In *USENIX Security Symposium*, pages 257–272, 2003.

[88] D. A. Ramos and D. Engler. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *Proc. USENIX Security Symposium*, 2015.

[89] D. A. Ramos and D. R. Engler. Practical, Low-Effort Equivalence Verification of Real Code. In *Proc. Int. Conf. on Computer Aided Verification (CAV)*, 2011.

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

[91] G. Russello, A. B. Jimenez, H. Naderi, and W. van der Mark. Firedroid: Hardening security in almost-stock android. In *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC)*, pages 319–328, 2013.

[92] F. Sabahi and A. Movaghar. Intrusion detection: A survey. In *2008 Third International Conference on Systems and Networks Communications*, pages 23–26. IEEE, 2008.

[93] S. Scannell. ebpf fuzzer. `https://scannell.io/posts/ebpf-fuzzing/`, 2020.

[94] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 167–182, 2017.

[95] 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.

[96] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2016.

[97] K. Singh. [PATCH bpf-next v1 00/13] MAC and Audit policy using eBPF (KRSI). `https://lwn.net/ml/linux-kernel/20191220154208.15895-1-kpsingh@chromium.org/`, 2019.

[98] S. Smalley, C. Vance, and W. Salamon. Implementing selinux as a linux security module. *NAI Labs Report*, 1(43):139, 2001.

[99] P. Srivastava and M. Payer. Gramatron: Effective grammar-aware fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 244–256, 2021.

[100] A. Starovoitov. BPF syscall, maps, verifier, samples, llvm. `https://lwn.net/Articles/609433/`, 2014.

[101] A. Starovoitov and D. Borkmann. net: filter: rework/optimize internal BPF interpreter's instruction set. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=bd4cf0ed331a275e9bf5a49e6d0fd55dffc551b8`, 2014.

[102] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proc. Network and Distributed Systems Security Symposium (NDSS)*, 2016.

[103] J. V. Stoep. Android: Protecting the Kernel. In *Linux Security Summit (LSS)*, 2016.

[104] S. M. S. Talebi, Z. Yao, A. A. Sani, Z. Qian, and D. Austin. Undo Workarounds for Kernel Bugs. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2381–2398, 2021.

[105] E. Torlak and R. Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, 2013.

[106] J. Van Geffen, L. Nelson, I. Dillig, X. Wang, and E. Torlak. Synthesizing jit compilers for in-kernel dsls. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II*, pages 564–586. Springer, 2020.

[107] S. Veggalam, S. Rawat, I. Haller, and H. Bos. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *European Symposium on Research in Computer Security*, pages 581–601. Springer, 2016.

[108] H. Vishwanathan, M. Shachnai, S. Narayana, and S. Nagarakatte. 'semantics, verification, and efficient implementations for tristate numbers. *arXiv preprint arXiv:2105.05398*, 2021.

[109] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, 2002.

[110] D. Wang, Z. Zhang, H. Zhang, Z. Qian, S. V. Krishnamurthy, and N. Abu-Ghazaleh. SyzVegas: Beating Kernel Fuzzing Odds with Reinforcement Learning. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2741–2758, 2021.

[111] 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 *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 612–624, 2017.

[112] X. Wang, D. Lazar, N. Zeldovich, A. Chlipala, and Z. Tatlock. Jitk: a Trustworthy In-Kernel Interpreter Infrastructure. In *Proc. USENIX OSDI*, 2014.

[113] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting Intrusions Using System Calls: Alternative Data Models. In *1999 IEEE Symposium on Security and Privacy (S&P). IEEE*, pages 133–145, 1999.

[114] F. Wei, S. Roy, X. Ou, and Robby. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proc. ACM CCS*, 2014.

[115] C. Welch. Pimax opens preorders for its very expensive 8K and 5K VR headsets. `https://www.theverge.com/2018/10/24/18019254/pimax-8k-5k-vr-headset-preorders-now-available-features-price`, 2018.

[116] M. Xu, S. Kashyap, H. Zhao, and T. Kim. KRACE: Data race fuzzing for kernel file systems. In *2020 IEEE Symposium on Security and Privacy (S&P). IEEE*, 2020.

[117] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 283–294, 2011.

[118] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proc. USENIX Security Symposium*, 2018.

[119] 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.

[120] Z. Zhang, Y. Cheng, S. Nepal, D. Liu, Q. Shen, and F. Rabhi. KASR: A reliable and practical approach to attack surface reduction of commodity os kernels. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 691–710. Springer, 2018.