UNIVERSITY OF CALIFORNIA,
IRVINE

Tackling Security Challenges in Operating Systems Using Selective Symbolic Execution

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Yingtong Liu

Dissertation Committee:
Professor Ardalan Amiri Sani, Chair
Professor Tuba Yavuz
Professor Alfred Chen

2022

DEDICATION

To God and my dearest life-long friends at UCI

TABLE OF CONTENTS

Page

LIST OF FIGURES **v**

LIST OF TABLES **vi**

ACKNOWLEDGMENTS **vii**

VITA **viii**

ABSTRACT OF THE DISSERTATION **x**

1 Introduction **1**
1.1 SSE for Programs with Untamed Environment 3
1.2 SSE for Reproducing Kernel Race Condition Bugs 4

2 Mousse: A System for Selective Symbolic Execution of Programs with Untamed Environments **7**
2.1 Background & Motivation 10
2.1.1 Selective Symbolic Execution 10
2.1.2 Program Environment 12
2.2 Design 14
2.3 Process-Level SSE 17
2.3.1 Memory Virtualization 20
2.3.2 Concretization Strategies 21
2.4 Environment-Aware Path Concurrency 22
2.5 Path Offloading & Distributed Execution 26
2.5.1 Path Offloading 27
2.5.2 Environment-Forced Symbolic Variables 30
2.6 Analysis 31
2.6.1 Android I/O Services 31
2.6.2 Target Analyses 32
2.7 Implementation 34
2.8 Evaluation 36
2.8.1 Performance 37
2.8.2 Coverage 41
2.8.3 Analysis Results 43

3 Macaron: Automatically and Reliably Reproducing Non-deterministic Race Condition Bugs without Reproducers in Syzbot **44**
3.1 Background . . . 47
3.1.1 Syzbot . . . 47
3.1.2 SIFT . . . 48
3.2 Syzbot Bug Study & Motivation . . . 49
3.3 Design . . . 50
3.4 Guided Selective Symbolic Execution . . . 52
3.4.1 Illustration Example . . . 53
3.4.2 Distance-based Interleaving Points Prioritizing Algorithm . . . 54
3.5 Multi-threaded Program Dependency Graph . . . 58
3.6 Implementation . . . 60
3.6.1 The Test Program . . . 61
3.6.2 Kernel Instrumentation . . . 61
3.6.3 IP Recording Optimization . . . 62
3.6.4 Threads Controlling . . . 63
3.6.5 MPDG for Kernel Modules . . . 64
3.6.6 IP Matching . . . 64
3.7 Evaluation . . . 66
3.7.1 Bug Case Study . . . 66
3.7.2 Experiments . . . 66
3.8 Discussions . . . 68
3.8.1 Reproducer . . . 68
3.8.2 Limitation . . . 69

4 Related works **71**
4.1 Symbolic and Concolic Execution . . . 71
4.2 Handle External Environment Interactions in Dynamic Testing . . . 73
4.3 Kernel Fuzzing in General . . . 76
4.4 Kernel Race Condition Bugs Detection . . . 77
4.5 Concurrency Bugs Reproducing and Debugging . . . 79
4.6 Other Related Work . . . 80

Bibliography **82**

LIST OF FIGURES

Page

2.1 *Simple hypothetical program used to describe the inner workings of SSE.* . . . 11
2.2 *Mousse design.* . . . 14
2.3 *Different SSE designs.* . . . 15
2.4 *A real code example demonstrating the importance of environment-aware concurrency. We have modified and eliminated parts of the code for clarity.* . . . 23
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.* . . . 24
2.6 *Simple hypothetical program used to demonstrate the offloading strategy in Mousse.* . . . 29
2.7 *Impact of environment-aware concurrency on execution time.* . . . 37
2.8 *Impact of distributed execution on execution time.* . . . 39
2.9 *Code coverage for different APIs of Android I/O services. Conc., M-I, and M-II refer to concrete execution and Mousse with concretization strategies I and II (§2.3.2), respectively.* . . . 42

3.1 *SIFT IP Illustration Example* . . . 49
3.2 *Multi-threaded GSSE Illustration Example* . . . 55
3.3 *First round discovered IPs of the illustration example* . . . 56
3.4 *Second round chosenIP set of the illustration example* . . . 56
3.5 *Third round chosenIP set of the illustration example* . . . 57
3.6 *Final round chosenIP set of the illustration example* . . . 57
3.7 *Algorithm 1: The main algorithm to update chosenIP set* . . . 57
3.8 *Algorithm 2: The threads interleaving points scheduling based on their weights* 58
3.9 *Illustration example's MPDG.* . . . 60
3.10 *Macaron workflow* . . . 65
3.11 *Example reproducer with semaphores* . . . 70

LIST OF TABLES

Page

2.1 *Single-API testing of OS services with Mousse. Abbreviations used in the table: GS = GPU Stack, AP = AudioProvider, CS = CameraServer, Res. = Resource constraint, Env. = Environment consistency.* 40

3.1 *Syzbot Bug Study* 50
3.2 *Bug Case Study* 68
3.3 *Different IP trace starting point evaluation* 68
3.4 *Different kernel bitcode size evaluation* 69
3.5 *PDG vs MPDG evaluation* 69

ACKNOWLEDGMENTS

I would like to express my sincere thanks to my advisor, professor Ardalan Amiri Sani. I am forever grateful for what I have learned from him, not only on how to solve challenging research problems but also anything about work and life attitude. Thank him for his patience and offering me the second chance when I wanted to give up. I know I could not have made it to the end of this journey without his encouragement and help during my Ph.D. study.

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

Hsin-Wei Hung is a wonderful workmate and friend. He contributed tremendously in the Mousse project. I thank him for all his support during the two years we worked together on the Mousse project. I thank Zhiyun Qian, Tuba Yavuz, Hsin-Wei Hung and Seyed Mohammadjavad Seyed Talebi on providing their insightful suggestions on the Macaron project.

I would also like to thank all my wonderful friends at UCI. I could have said all the wonderful memories I had with them and made that very long. But I just want to list their names here. Because both they and I know we will have even more wonderful times together in the future. I thank Zhihao (Zephyr) Yao, Sichao Liu, Xiaoshuang (Iris) Luo, Lily Wu, Linyi Xia, Harry Xu, Becky Chen, WenKai (Kevin) Zhang, Zhe Wang, William (Bill) Heidbrink and his wife Mary.

Thank my parents and my dear sister, for loving and supporting me unconditionally.

Finally, I would like to acknowledge the funding support by the National Science Foundation (NSF) Awards #1763172, #1617513, and #1617481.

VITA

Yingtong Liu

EDUCATION

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

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

Bachelor of Computer Science and Engineering **2015**
Huazhong University of Science and Technology *Wuhan, China*

RESEARCH EXPERIENCE

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

TEACHING EXPERIENCE

Teaching Assistant **2017–2018**
University of California, Irvine *Irvine, California*

REFEREED CONFERENCE PUBLICATIONS

Sifter: Protecting Security-Critical Kernel Modules in Android through Attack Surface Reduction. **October 2022**
Proc. ACM Int. Conf. Mobile Computing and Networking (MobiCom)

Mousse: A System for Selective Symbolic Execution of Programs with Untamed Environments. **April 2020**
Proc. ACM European Conference on Computer Systems (EuroSys)

Sugar: Secure GPU Acceleration in Web Browsers. **March 2018**
Proc. ACM Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)

ABSTRACT OF THE DISSERTATION

Tackling Security Challenges in Operating Systems Using Selective Symbolic Execution

By

Yingtong Liu

Doctor of Philosophy in Computer Science

University of California, Irvine, 2022

Professor Ardalan Amiri Sani, Chair

Symbolic execution is a widely used program analysis technique to systematically execute different program execution paths without requiring concrete inputs. Developed from symbolic execution, Selective Symbolic Execution (SSE) is a technique for creating the illusion of full system symbolic execution, while symbolically running only the code that is of interest to the analyst. SSE makes symbolic execution practical for large software like an operating system by alleviating the problem of path explosion.

In this thesis, we demonstrate the possibility of using SSE to tackle security challenges in operating systems. Our first work is Mousse, which is a system for selective symbolic execution of programs with untamed environments. Mousse facilitates testing operating system services that are highly coupled with their underlying environment by developing three novel solutions. Our second work is Macaron, which is the first framework that can automatically and reliably reproduce non-deterministic race condition bugs in the operating system kernel. We demonstrate the usefulness of Macaron by using it to reproduce bugs found (but not reproduced) by syzbot, the state-of-the-art continuous kernel fuzzing framework. We build Macaron using a novel Guided Selective Symbolic Execution (GSSE), static analysis, and a distance-based thread interleaving points scheduling algorithm.

Chapter 1

Introduction

The Operating System (OS) provides the fundamental mechanisms to support all kinds of applications on computing devices like personal laptops and smartphones. It powers almost every computing device in our daily life. As the bridge between applications and the hardware, a wide range of functionalities and features are incorporated into the OS to support the ever increasing needs of different applications. From the user space OS services to the device drivers in the kernel, OS exposes a large Trusted Computing Base (TCB) and has became a hot target for attackers ever since its presence. Any vulnerability in the OS could potentially cause the attackers to compromise the applications or the system itself. For example, about 120 new CVEs were reported in the Linux kernel just in 2020 according to our search in the MITRE CVE database [7]. A recent study [110] also shows that the number of attacks to operating systems are growing rapidly. Attacks and exploits to the OS not only impact our daily use of the devices but also can cause a lot of companies to lose millions of dollars due to important information leakage.

The research community has explored multiple ways to improve the security of the OS. Isolation [54, 92, 121, 11] is a commonly used technique to isolate the vulnerable OS components

from the rest of them. Formal verification can be used to formally verify a sub-component of the OS [86]. Rust [8] as a Type-safed language has became very popular and was proposed to replace part of the C code in the kernel. ARM TrustZone [1] and Hardware virtualization are two hardware protection techniques that can provide the hardware isolation for critical components of the OS. Traditional program analysis techniques like static analysis [21], fuzzing [107] and symbolic execution [45] have also been intensely used to test the robustness of the OS. Among all those program analysis techniques, symbolic execution has gained popularity since 2005. It was pioneered by a lot of practical tools [22, 34, 30, 69, 118, 112]. The idea of symbolic execution is to systematically execute different program execution paths without requiring concrete inputs. It greatly reduces redundant executions by uniquely identifying each execution path with a set of constraints on the inputs when exploring different execution paths in a program. However, OS as one of the most complicated software system faces a lot of challenges when using symbolic execution. As many works [44, 93, 95] demonstrate, directly applying symbolic execution on the OS causes severe path explosion problem and the failure of complex constraints solving.

To make symbolic execution practical and applicable to the OS, developed from symbolic execution, Selective Symbolic Execution (SSE) was proposed by the S2E [44] team. SSE allows symbolic execution to scale to real-world programs with in-vivo analysis, which means analyzing the program with its real living environment present. It is a technique for creating the illusion of full system symbolic execution, while symbolically running only the code that is of interest to the analyst. SSE cleverly maintains a concrete state and a symbolic state at the same time and allows back-and-forth switching between them. Because of this, usually the analyst can initialize an OS in S2E's concrete execution and switches to symbolic execution as needed when the tested code is being executed. S2E [44] demonstrates SSE's use case in developing practical tools for comprehensive performance profiling, reverse engineering of proprietary software, and bug finding for both kernel-mode and user-mode binaries.

In this thesis, we demonstrate two practical applications of SSE to tackle some security challenges specifically in the operating systems. In Chapter §2, we show how we use SSE for programs with untamed environments. To do so, we introduce Mousse, the first framework that addresses the challenges in testing programs with environments that cannot be modeled nor virtualized in symbolic execution. It opens the opportunity to do flexible analysis for an important set of programs in customized operating systems, especially for mobile devices. In Chapter §3, we introduce Macaron, a first-of-its-kind framework that applies SSE for reproducing kernel race condition bugs found by syzbot [9].

1.1 SSE for Programs with Untamed Environment

Dealing with complex external environments that the tested programs interact with has always been a challenge in symbolic execution. The symbolic execution engine represents the program's execution path by adding constraints on each path, assuming it has full control over the program's behavior. However, a lot of real-world programs interact with external environments heavily. When the program interacts with external environments, the symbolic execution engine loses its track of the program's behaviors and thus cannot do an accurate analysis of the program. For example, a network agent interacts with an external server to send or receive network packages. Such external environments are usually out of the control of the symbolic engine.

A common way to deal with such environments is to model the environments by redirecting calls that access it to models that understand the semantics of the desired action well enough to generate the required constraints. One example is KLEE [33], which models the system calls to deal with programs issuing system calls. Modeling the external environment plays an important role in symbolic execution, but it has its own problem. The accuracy of the analysis heavily depends on how accurate the provided models are. In some cases,

the models are too complex to model and hence they result in false positives in symbolic execution. Another way is to use selective symbolic execution with the full virtualization of the whole system like what S2E does. SSE in S2E conceptually gains full control over the whole guest OS by using virtualization. In its design, the programs running in the guest operating systems can be thoroughly analyzed since SSE understands the semantics of the whole OS that the programs interact with.

However, for programs that interact with environments that cannot be virtualized nor modeled, S2E is not applicable. We give such programs a special name: programs with untamed environments. Such programs include OS services managing I/O devices, software frameworks for accelerators, and specialized applications. Since we would like to enable the use of SSE for more and more real-world programs, it is necessary to solve the challenge of using SSE for programs with untamed environments.

In §2, we present a framework called Mousse that addresses this critical challenge, that is the applicability of SSE to a large and important set of programs: programs with untamed environments. We tackle this challenge with three solutions. These include a novel process-level SSE design, environment-aware concurrent execution, and distributed execution of program paths. We use Mousse to comprehensively analyze five OS services in three smartphones. We perform bug and vulnerability detection, taint analysis, and performance profiling. Our evaluation shows that Mousse outperforms alternative solutions in terms of performance and coverage.

1.2 SSE for Reproducing Kernel Race Condition Bugs

The kernel, as one of the most complex concurrent systems, is implemented with fine-grained concurrency mechanisms in mind to fully utilize the power of multi-core hardware. Spin-

locks [6] and mutexes [3] are the two basic mechanisms used to protect critical sections in the kernel. Even more complex concurrency mechanisms include semaphores [5] and Read-Copy-Update (RCU) [4]. Because of the kernel's concurrent and non-deterministic nature, race conditions bugs in the kernel are pervasive and constantly being discovered by the kernel fuzzers. Many research works [52, 113, 108, 111, 55, 74, 65, 120, 39, 67] have put great effort into detecting race condition bugs more effectively, mainly by kernel fuzzing [65, 120, 39, 67]. Fuzzing [81, 28, 107] is very efficient in detecting bugs in the kernel because it can generate massive test cases and run the testing continuously. But fuzzing itself has a fundamental problem in providing useful debugging information for the bugs it finds. In many cases, bugs that cannot be reproduced are found by fuzzers. Because of lacking a way to reproduce the bug, the analysts cannot understand the root cause of the bug. This is especially true for kernel race condition bugs without reproducers. Race condition bugs may be triggered randomly when the system performs a specific interleaving of the threads. However, many fuzzers cannot recreate the exact interleaving of the threads after a non-deterministic bug is triggered. To reproduce and further fix a race condition bug, knowing the exact thread scheduling is critical.

Realizing this problem in the kernel fuzzing world, we resort to SSE for solutions to automatically and reliably reproduce long-existing kernel race condition bugs without reproducers. Specifically, we focus on the bugs reported by syzbot [9], the state-of-the-art continuous kernel fuzzing platform. SSE is extremely helpful in analyze a program's execution path and guarantees the program can run correctly without false positives at the same time. Our goal is to reproduce a non-deterministic race condition bug by satisfying four requirements: 1. Automatic 2. Reliable 3. Efficient 4. Compatible. We think to reproduce a race condition bug reliably, false positives should be eliminated. Compared to static analysis, SSE is much more suitable to tackle kernel race condition bugs with its reliable execution and delicate analysis of each execution path. To reproduce a bug as soon as possible, in other words, to make the reproducing process efficient, we propose Guided Selective Symbolic Execution

(GSSE).

In §3, we introduce our framework called Macaron, which uses GSSE for reproducing kernel race condition bugs. We use all the information provided in syzbot to identify the critical features of a bug and optimize SSE's search space in finding a specific thread interleaving that manifests the bug. Except for SSE, Macaron uses static analysis and a distance-based interleaving points prioritizing algorithm to reproduce race condition bugs without any modifications to the current design of syzbot.

Chapter 2

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

Selective symbolic execution 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 [73, 44, 95], performance profiling [44] and reverse-engineering of binaries [42, 44]. In addition, it can be used for taint analysis, hybrid fuzzing [105, 125], and for exploit generation and analysis [19, 20].

In this work, 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 [101, 19, 20, 38] and virtualizing it [44]. Unfortunately, neither approach is feasible for *untamed* environments, i.e., those that include diverse hardware components and their device drivers. Examples include OS 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 [73, 42, 95], 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 [126, 82, 13]. 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 OS process), which integrates the symbolic and concrete execution engines in the same OS 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 OS 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 OS 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 single device time 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 OS 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 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 OS 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 [14].

2.1 Background & Motivation

2.1.1 Selective Symbolic Execution

SSE is a powerful program analysis technique that can analyze multiple execution paths of a program [43, 44, 45, 126, 82, 13, 38, 77]. 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 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

```
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.*

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., OS process fork, and is discussed in §2.3. 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 (see §2.3 for more detail).

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.

2.1.2 Program Environment

The environment of a program is the set of all hardware and software components that it interacts with. This includes the OS 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 [101, 19, 20, 38], 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 [44].

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 [12] or an in-display fingerprint scanner [29]; a voice assistant may employ arrays of speakers and microphones for audio beamforming [88]; and a VR/AR headset may employ high-resolution displays, requiring powerful GPUs [117]. 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 [37]. The programs that interact with these devices include OS 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, 75, 59, 26, 76], is not enough, as it does create multiple virtual instances of the device.

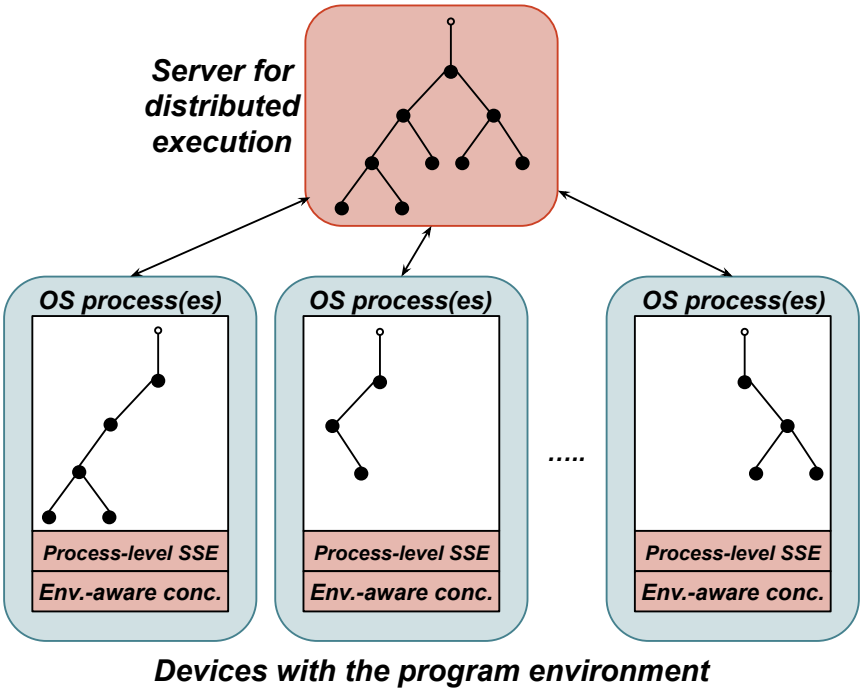


Figure 2.2: *Mousse design.*

2.2 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.

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

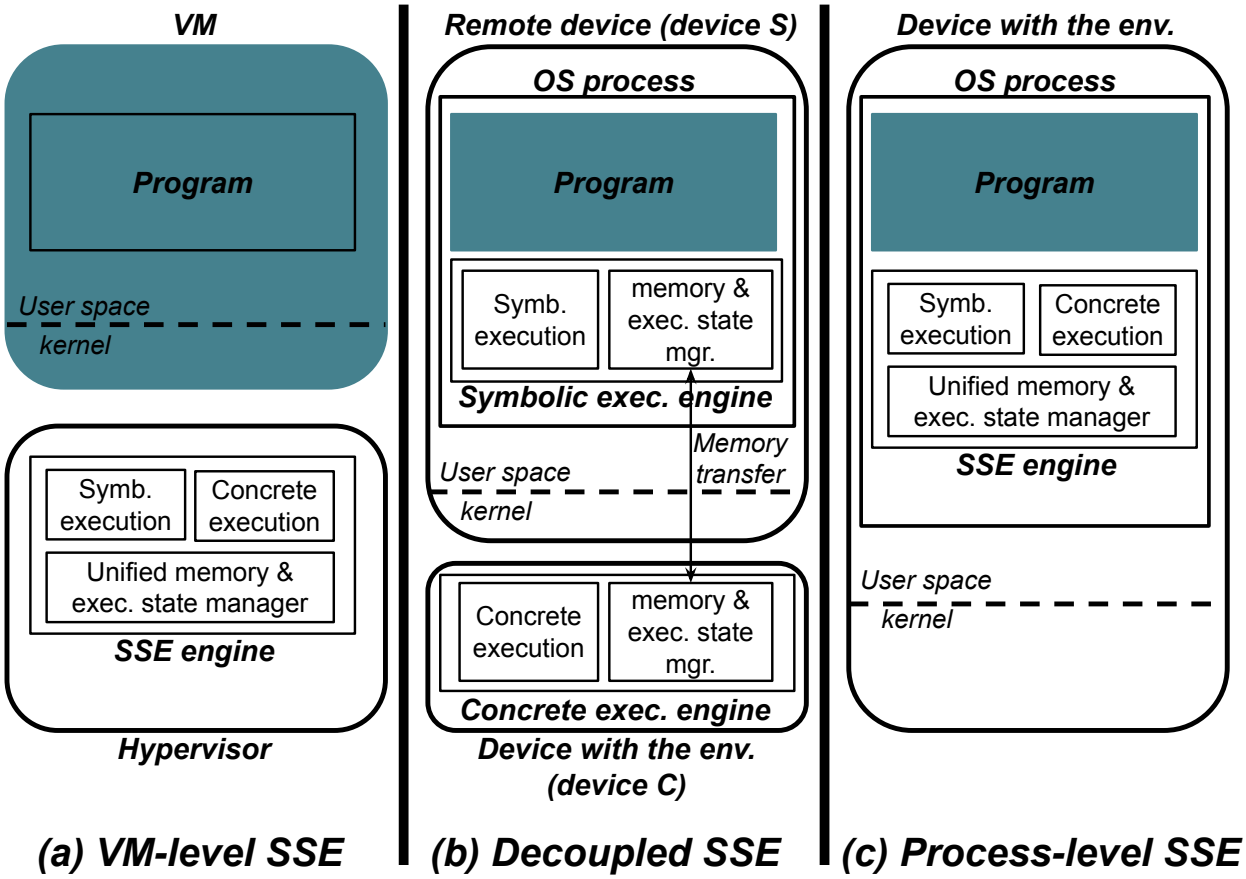


Figure 2.3: *Different SSE designs.*

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 approach 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.

2.3 Process-Level SSE

In this section, we describe the *process-level SSE* design used in Mousse. Our key contribution is to run both symbolic and concrete execution engines in the OS process that contains the program itself. We describe existing SSE designs and their shortcomings, before presenting more details on our design.

Existing SSE designs. To tackle the issue of applying SSE to a program with untamed environment, the first design that one might consider is *VM-level SSE*, as implemented in S^2E [44, 45]. Figure 2.3 (a) shows the design of VM-level SSE. To use it, the analyst runs the program in a VM. The symbolic and concrete execution engines are implemented within the hypervisor and share a unified memory and execution state. When a program path needs to be explored, the SSE engine forks the whole VM, giving each program a completely separate environment.

Unfortunately, using VM-level SSE for analyzing programs with untamed environments is generally not feasible, since virtualization of the hardware component in the program's environment (needed to run the program in a VM) is generally not possible (§2.1.2). For example, we are not aware of a solution that can virtualize the various I/O devices of a smartphone.

The second design one might consider is *decoupled SSE*, as implemented in Avatar [126], Avatar^2 [82], and Symbion [13]. In these systems, the concrete execution engine is configured to directly run on top of the program's environment. The symbolic execution engine runs elsewhere, e.g., in a server or workstation, and communicates with the concrete execution engine remotely. Unlike VM-level SSE, the decoupled SSE design is capable of analyzing programs with untamed environments.

Figure 2.3 (b) shows the design of a decoupled SSE. We illustrate two devices, C and S.

Device C has the program's environment and a concrete execution engine. Device S does not have the environment but has a symbolic execution engine. The system starts the symbolic execution in Device S, and transfers execution to C when the environment is needed. In this case, as the symbolic and concrete execution engines are in separate devices, they have to transfer the memory state when switching the execution mode.

Unfortunately, as our experience shows, decoupled SSE has two drawbacks. The first is the performance overhead of transferring memory state between the symbolic and concrete execution engines. In §2.8.1, we evaluate this overhead using Avatar^2, and show that process-level SSE reduces execution time by at least 63%. The second is that it is hard to configure and use. This is because one needs to set up the symbolic and concrete execution engines separately and configure the memory state transfer channel between them.

A final option that one might consider is to use a symbolic environment, where all return variables from the environment are marked as symbolic, and hence the real environment is not needed. This is the approach used in DDT [73], RevNIC [42], and SymDrive [95]. Unfortunately, as we will report, this approach significantly increases the number of symbolic variables and hence results in path explosion as well as false coverage. We tested this approach on three Android I/O services. None of the services initialized correctly even after a few days of execution. We do, however, note that a better path scheduling algorithm, similar to the ones used by SymDrive [95], could potentially alleviate the effect of path explosion, but we did not explore that.

Process-level SSE. These challenges prompted us to design and build a new SSE approach, which we call *process-level SSE*. In this design, both the symbolic and concrete execution engines run within the same OS process that hosts the program. To analyze a program, one loads the SSE engine into a process and have the engine load and execute the program. Thus, both the symbolic and concrete execution engines can easily interact with the environment. In the rest of the paper, we refer to the interactions of the program with the environment

as *environment calls (ecalls)*. Whenever the program issues an ecall (either in concrete or symbolic modes), the SSE engine passes it to the underlying environment for execution.

Figure 2.3 (c) shows the design of process-level SSE. Both engines are in the same process as the program, which is located in the device with the environment of interest. The two engines, similar to VM-level SSE, use a unified memory and execution state and enable the program to interact with its environment.

Process-level SSE supports concurrent execution of program paths. To achieve this, it executes each program path in a separate OS 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 thanks to copy-on-write support in the kernel.

One key benefit of this design (compared to decoupled SSE) is improved performance. As both engines share memory, this eliminates the need to transfer memory state. The other benefit is that process-level SSE is easier to use than counterparts. Analyzing a program with VM-level SSE requires launching a VM and running the program in the VM. Analyzing a program with decoupled SSE requires configuring the symbolic and concrete execution engines in two separate devices and configuring a channel for memory state transfer. In contrast, in Mousse, the analyst only needs to load the SSE engine and the program in an OS process, which can be done with a single command in the OS shell.

Process-level SSE has its own limitations. First, since the engines execute in the same process as the program under analysis, the device must have adequate computing power. Process-level SSE is best suited to high-end mobile devices (such as smartphones, tablets, and laptops) as well as desktops and servers. Decoupled SSE is the right design for weak devices, e.g., embedded devices. Moreover, process-level SSE cannot analyze the OS kernel code, nor programs with multiple processes. VM-level SSE is the right design in these cases.

2.3.1 Memory Virtualization

As mentioned, an SSE engine emulates the instructions in a program. In doing so, it virtualizes the process address space for the program. Therefore, the address space seen by the program (i.e., guest address space) could be different from that of the OS process that it runs inside (i.e., host address space). This way, the memory used by the program is isolated from the memory used by the SSE engine itself. This virtualization requires the SSE engine to maintain the mapping between the guest and host address spaces and to translate when emulating memory-access instructions.

However, the native execution of syscalls creates a challenge for address space virtualization. That is, addresses passed to the kernel through the syscall arguments are in the guest address space, whereas the kernel uses the host address space to dereference the memory pointers passed to it. Unfortunately, simply translating the addresses in the syscall arguments is not enough. This is because the data buffers passed to the kernel must be contiguous in the host address space. This is not necessarily the case, since the program allocates these buffers in the guest address space.

To address this problem, we configure the guest addresses to be identical to their underlying host addresses. This way, if a buffer is contiguous in the guest address space, it is contiguous in the host address space too. The limitation of this approach is that those addresses used by the SSE engine in the host address space cannot be used in the guest address space. However, given the large set of addresses available in an address space in modern ISAs, this limitation is not serious in practice.

2.3.2 Concretization Strategies

Since the environment cannot be modeled nor virtualized, any symbolic arguments passed to ecalls must be concretized. In this section, we present two different concretization strategies supported by Mousse.

Strategy I: constrained concretization. In this strategy, the symbolic arguments passed to ecalls are concretized and the execution path is constrained. That is, Mousse chooses a concrete value for the symbolic argument that satisfies all the path constraints, and adds a new constraint to the path enforcing the value of the variable to be equal to the concrete value. With this strategy, the ecall returns a concrete value.

This strategy results in no false positives since all executed paths are correct program paths. However, this strategy can potentially limit the coverage and the number of paths explored due to the additional constraints and the concrete values of the outputs of ecalls. This limitation happens only when an argument to an ecall is symbolic, forcing Mousse to concretize it. Fortunately, as our experiments show, ecalls with symbolic arguments are rare in OS services that we analyze. In other words, the service inputs marked as symbolic in the analysis rarely propagate to ecall arguments. The only such case that we have noticed are when OS services log the program inputs to the terminal by using a `writev` syscall. To avoid these, the analyst can disable the logging in the service.

Strategy II: concretization with unconstrained input and symbolic output. In case a program does have ecalls with symbolic arguments, constrained concretization limits the coverage. To address this issue, Mousse provides a second strategy, in which it takes two actions. First, when a concrete value for the symbolic argument is chosen, it does not add the corresponding constraint to the path. Second, it marks the outputs of these ecalls as unconstrained symbolic variables, hence allowing the forking and execution of paths that depend on the values of the outputs of these ecalls.

Note that while this approach may result in false coverage, it forks fewer paths and produces less false coverage compared to the symbolic environment approach discussed earlier, as we will show empirically in §2.8.2. This is because the latter marks the outputs of all ecalls as symbolic, whereas the former marks only the outputs of ecalls with symbolic arguments as symbolic.

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 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

```
/* 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.*

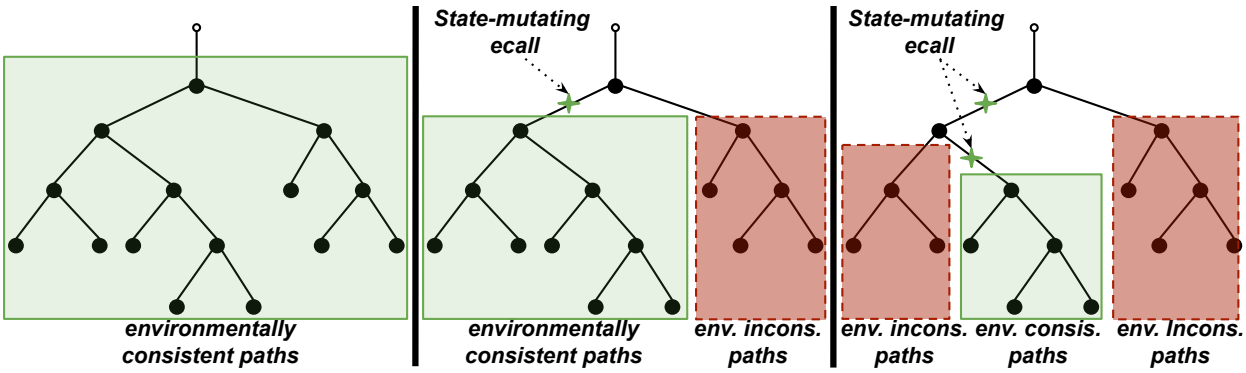


Figure 2.5: *Environment consistency for concurrent path execution. (Left) All paths to be executed in a device. (Middle) Environmentally consistent and inconsistent paths after a state-mutating ecall. (Right) Paths after the second state-mutating ecall.*

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 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, 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 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

```
 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.*

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 (Strategy II in §2.3.2) 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 paper, 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 [80]). 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 [44, 45], 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 [116], and DroidSafe [60]) 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 [44, 45], QEMU (user-mode execution) [24], and KLEE [33]. 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 OS. 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) [25] 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 OS 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`.

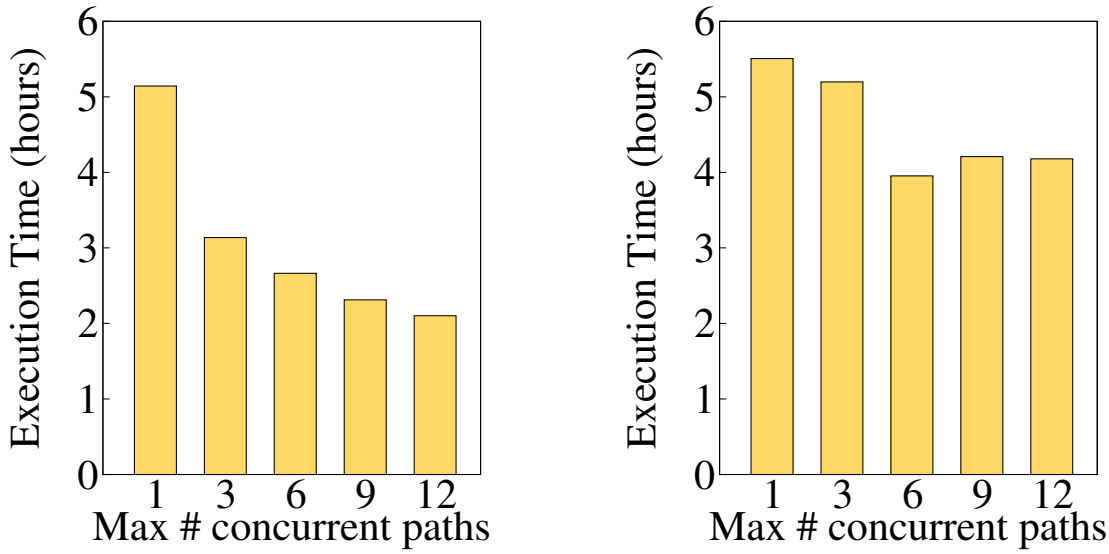
We do not currently support signals, as none of the programs we have analyzed use signals from the environment, e.g., from the driver. Instead, these programs use syscalls (such as `poll` and `select`) to receive notifications. We do, however, support per-process signals, such as `SIGTERM`.

Syscall inputs and outputs. Mousse needs to correctly identify all inputs and outputs of syscalls. It needs to know the inputs for concretization. It needs to know the outputs to mark them as symbolic in concretization strategy II (§2.3.2). Implementing this is challenging since syscall inputs and outputs may contain untyped pointers. One important syscall that exhibits this behavior is the `ioctl` syscall, which receives three arguments `(struct file *file, long cmd, void *arg)`. The type of the third argument depends on the value of the second one. This syscall is used heavily by device drivers, and hence is called frequently in Android I/O services.

To address this issue, Mousse needs to know the type of these pointers. We manually extract the type information from the header files in a driver source code and include it inside Mousse's source code.

2.8 Evaluation

We evaluate three aspects of Mousse: performance, code coverage, and analysis results. In our evaluation, we use five OS 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).



(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.*

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 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. Finally, we 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%.

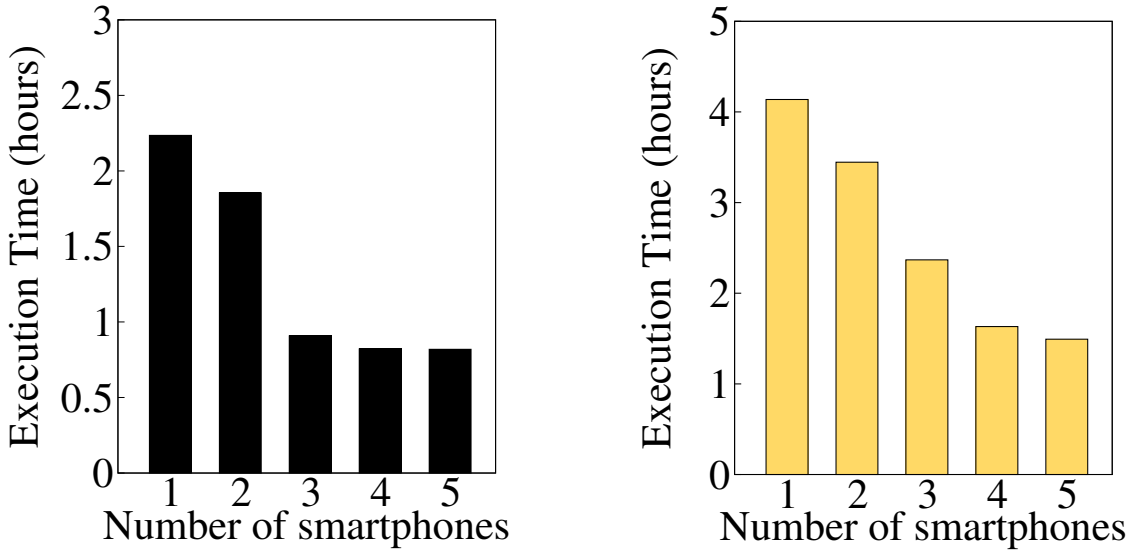
Environment-aware concurrency. Figure 2.7 shows the execution time of two APIs of the AudioProvider in Pixel 3 when varying the maximum number of concurrent paths allowed on the device. The figure shows significant benefit from concurrency for one API and modest benefit for the other. This is due to 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 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 OS 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



(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.*

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.

Comparison with decoupled SSE. We compare the performance of Mousse's process-level SSE design with the state-of-the-art decoupled SSE solution, Avatar^2 [82]. We note that Avatar^2 does not support concurrent execution of program paths interacting with the environment. It does not support distributed execution either. Therefore, we only compare the performance of a single path execution using a single smartphone.

We use Avatar^2 to test one API of the AudioProvider service, `adev_open` in Pixel 3. We run the symbolic execution engine of Avatar^2 in an x86 server, run the concrete execution engine in a Pixel 3 smartphone, and have them communicate using Android Debug Bridge (ADB). Avatar^2 uses GDB for its concrete execution engine. We start with concrete execution on the smartphone. We use GDB to set a breakpoint right before the call to `adev_open`. Then, we switch the execution from concrete mode on the smartphone to symbolic mode on the server. We also set two breakpoints after the switch to measure the execution time from the

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 OS services with Mousse. Abbreviations used in the table: GS = GPU Stack, AP = AudioProvider, CS = CameraServer, Res. = Resource constraint, Env. = Environment consistency.*

switch to the time the execution reaches the breakpoints. After the switch, Avatar^2 reads the memory of the concrete execution engine over ADB to synchronize the state of the symbolic execution engine.

Avatar^2 took 24.86 seconds to initialize the AudioProvider all in concrete mode. However, it then took 257.47 seconds to switch the execution mode, read the remote memory, and reach the first breakpoint in symbolic mode. Unfortunately, Avatar^2 could not reach the second breakpoint. More specifically, Avatar^2 was aborted due to a "read-miss" error after running

for another 1 hour and 44 minutes.

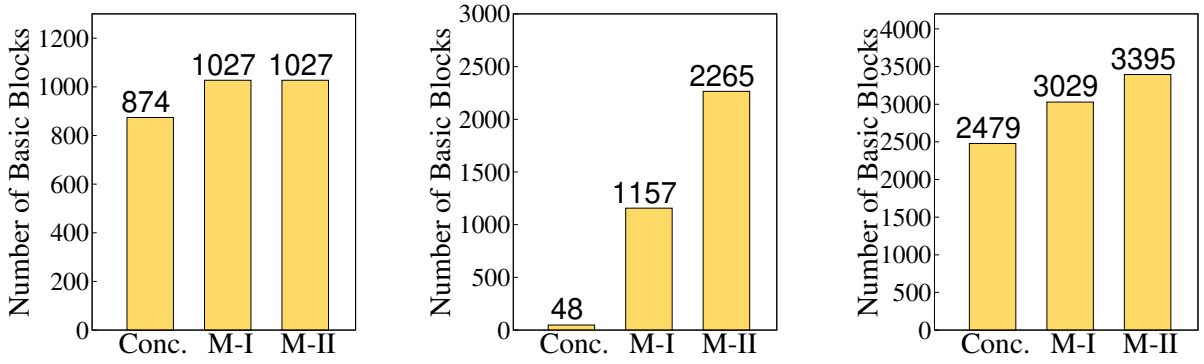
As a comparison, using Mousse with one phone executing one path at a time (i.e., no concurrency), we were able to finish testing a path of `adev_open` completely in 104 seconds. Mousse took 40 seconds to initialize the AudioProvider service, which is slower than Avatar^2. This is because Mousse's concrete execution engine, based on QEMU, fully emulates all instructions. However, Mousse's unified memory avoids costly memory transfers, allowing it to significantly outperform Avatar^2. Compared to Avatar^2, which took 282.33 seconds to reach the first breakpoint after the switch, Mousse improves performance by at least 63% as it finishes the whole path in 104 seconds.

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 with its two concretization strategies (§2.3.2). The figure shows two important points. First, it shows that Mousse achieves better coverage than concrete execution. Second, it shows that, in the absence of



(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., M-I, and M-II refer to concrete execution and Mousse with concretization strategies I and II (§2.3.2), respectively.*

syscalls with symbolic arguments, both concretization strategies in Mousse achieve the same coverage (Figure 2.9a). Syscalls with symbolic arguments are rare in Android I/O services that we have analyzed. The only such syscalls are those for logging, as discussed in §2.3.2, which one can disable before analysis. However, in the presence of such syscalls, the second concretization strategy achieves higher coverage (Figures 2.9b and 2.9c). But we note that it is not known how much of this is false coverage, i.e., execution that would not occur in normal execution. Determining how much requires further analysis.

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 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.

Chapter 3

Macaron: Automatically and Reliably Reproducing Non-deterministic Race Condition Bugs without Reproducers in Syzbot

Concurrent systems are pervasive due to the increase of advanced multi-core hardware. The kernel, as one of the most complex concurrent systems, is implemented with fine-grained concurrency mechanisms in mind to fully utilize the power of multi-core hardware. While the concurrency optimizations in the kernel greatly speed up the whole system's execution, they make it notoriously prone to race condition bugs. Those bugs are hard to find during testing due to the requirement of specific thread scheduling. Even worse, they are hard to reproduce once found due to the non-deterministic nature of the kernel. Kernel race condition bugs can have a very serious impact on users and the system itself, such as deadlocks, kernel panics, data corruptions, and even privilege escalation attacks.

In response to these issues caused by race conditions, many research works have put a great amount of effort into detecting race condition bugs more effectively. Those works have a common agreement that to detect a race condition bug, two design requirements need to be satisfied: 1. Find a test program that may execute potential race locations in different threads of the system. 2. Systematically control the thread interleaving in the system to manifest a race condition. To generate valid test programs that can be invoked against specific kernel modules, the most popular kernel fuzzer, syzkaller [2], provides templates that encode the detailed information regarding syscalls, including the type of arguments of each syscall and the dependencies of different syscalls. But it does not do anything to control the thread interleaving of the kernel. Being aware of this problem that syzkaller has in finding race condition bugs, other works [98, 89, 65, 70, 58] improved the efficiency of fuzzing by introducing systematic thread interleaving exploration and generating input programs that accesses different race locations suggested by static analysis. Among those works, including syzkaller, some race condition bugs are reproducible, but most of them are not because their main focus is still finding more race condition bugs.

However, for a race condition bug, whether it is reproducible or not plays a vital role in understanding the root cause of the bug and further fixing the bug. For concurrency bugs in user space programs, the software community has tried multiple resorts [79, 104, 127, 79] to reproduce them a long time ago, mainly by recording critical runtime information [104], analyzing offline information including core dumps [119, 27] or guessing problematic schedules based on good runs [79]. However, we found that a framework that can be used to reliably reproduce kernel race condition bugs is missing. The reason is that reproducing race condition bugs in the kernel is very challenging. The old methods like recording runtime interleaving information cannot be used for the kernel due to its unbearable overhead. Moreover, a thread controlling algorithm that can exhaust all the possible thread interleavings in the kernel is not possible even with an optimized design.

In this work, we propose a framework called Macaron to automatically and reliably reproduce race condition bugs without reproducers in syzbot [9]. Race condition bugs in syzbot can be reported as use-after-free, double-free, deadlocks, etc. Macaron does not modify syzbot's current implementation to record any additional runtime information. It only uses the existing offline information collected by syzkaller with static analysis, selective symbolic execution, and a distance-based thread interleaving points prioritizing algorithm to reproduce race condition bugs. In this work, we make the following contributions:

1. Macaron does not require any changes to syzbot. Therefore, we can tackle the old and long existing race condition bugs reported by syzbot. This is very important to improve the security of the kernel. Even though finding more new race condition bugs is important, fixing already discovered race condition bugs is equally important. A lot of discovered race condition bugs remain unresolved or marked as invalid because of lack of reproducers. Reproducers or Proof of Concept (PoC) programs are usually essential in helping analysts fix the bugs. Macaron is the first framework targeting finding reproducers for race condition bugs without reproducers in syzbot.

2. Macaron proposes a novel way called Guided Selective Symbolic Execution (GSSE) to reproduce race condition bugs by combining three main techniques, namely, program dependency analysis, SSE, and distance-based thread interleaving points prioritizing algorithm. We first use the offline information from syzbot to find a potential test program in its execution log. We then run the kernel with that test program using SSE. During the execution, we trace potential interleaving points in the kernel modules related to the bug. Finally, Macaron's SSE is guided by a distance-based thread interleaving points prioritizing algorithm to greatly reduce the exploration space.

3. As an initial try, we experiment Macaron on some use-after-free bugs without reproducers found by syzbot to evaluate its performance. The evaluation shows Macaron now can successfully reproduce bugs that have one race condition.

3.1 Background

3.1.1 Syzbot

Syzbot [9] is the state-of-the-art continuous kernel fuzzing framework. It continuously fuzzes main Linux kernel branches and automatically reports found bugs to kernel mailing lists. Syzbot uses Syzkaller, which is an unsupervised coverage-guided kernel fuzzer. It was developed with kernel fuzzing in mind. One of syzkaller's highlighted feature is its accurate syscall templates for a wide range of kernel modules. Using those templates, syzkaller generates input that understands the built-in patterns of the kernel module under testing. Those syscall templates are written by different people and are becoming more and more accurate. Benefiting from syzkaller's special templates and distributed continuous fuzzing, syzbot reports a lot of bugs in the kernel. Once a bug is found, its reproducer will automatically rerun all the test cases that have been tested until the point a bug is triggered to find a reproducer. The kernel developers then can use the reproducer provided along with the bug report to understand and investigate the bug. For bugs that cannot be reproduced by syzbot automatically, if no one submit a patch to fix the bug after a period of time (usually over a month), the bug will be auto closed.

For simple and deterministic bugs, the reproducer can be generated successfully. However, since kernel is highly non-deterministic, especially for race condition bugs, syzbot will fail to find a reproducer. Even if a reproducer sometimes is found for such non-deterministic bugs, users find they cannot reproduce the bug using the provided reproducer. Apparently, for non-deterministic race condition bugs, simply rerunning the test program or their permutations in the kernel is not enough. Thread interleaving control is also absolutely needed to reproduce such bugs. Macaron targets on this problem and does efficient thread interleaving exploration for the race condition bugs that can not be reproduced in syzbot.

3.1.2 SIFT

SIFT [122] is a property directed symbolic execution of multi-threaded user-space code. It can analyze an execution trace of a multi-threaded program to identify the program locations at which a thread Interleaving Point (IP), i.e., context switch, may lead to a failure of a correctness property such memory safety or an invariant that must hold at a specific program location. Such program locations are called interleaving points. SIFT's inference rules to infer interleaving points works in three steps for each symbolic execution path: 1. Collecting data-flow and control-flow facts about the instructions executed on that path, 2. creating thread access information for each property relevant object discovered in Step 1, and 3. filtering out accesses that must not happen in parallel.

For example, in a very simple example with two threads in Figure 3.1, SIFT can detect operations that access the global variable x and mark them as IPs. The first IP is when the program counter of the main thread refers to the read operation at line 9 and the second IP is when the program counter of the child thread refers to the write operation at line 3. Let us assume the child thread starts right after it is created. If after executing IP at line 3, a context switch happens from the child thread to the main thread at line 9, the assertion at line 10 can be triggered. Scheduling threads based on their IPs performs better than randomly doing thread schedulings in terms of finding bugs. SIFT provides a set of inference rules that can be used to infer IPs in a multi-threaded program using symbolic execution and works in an iterative way. In each round, it explores the state space using selective scheduling based on a set of thread interleaving points that have been inferred in the previous round. A round of execution stops when all the IP permutations have been explored and the next round will start exploring newly inferred IPs. It also optimizes the state space search by using three different IP partitioning strategies, with each one having different strengths. In a program with a large set of IPs, the state search space is huge and can take unlimited rounds. The whole execution of SIFT terminates when it reaches a timeout or

triggers a bug. In our work, we adopt part of SIFT's design. We use its inference rules to infer potential interleaving points that access the global memories or shared memories in the kernel during symbolic execution. As for the state space search, we do not use any IP partitioning strategies from SIFT. Instead, we propose GSSE to satisfy our specific requirements. GSSE will be explained in section §3.4.

```
1  int x;
2  void *accessorThread(void *arg){
3     x = -1;
4  }
5  int main(int argc, char *argv[]){
6    int res = 0;
7    pthread_t acc;
8    pthread_create(&acc,NULL,accessorThread,NULL);
9    if(x < 0)
10       assert(0);
11    pthread_join(acc,(void**)&res);
12    return 0;
13 }
```

Figure 3.1: *SIFT IP Illustration Example*

3.2 Syzbot Bug Study & Motivation

Before we design Macaron, we did a preliminary study on the bugs in syzbot to get an idea of how many of them do not have reproducers. We collected three data points in syzbot at different times, shown in table 3.1. Based on the data, the percentage of non-reproducible bugs does not change too much over three years, which is around 24 percent of all the bugs listed in syzbot. All those non-reproducible bugs can be further categorized into 12 types: general assertion hit, boot error, general warning hit, out of bounds, suspicious RCU usage, task hung, general protection fault, double free or invalid free, potential deadlock, uninitialized value, null-pointer dereference and use-after-free. Among all the 12 types, boot error and uninitialized value are less likely caused by race condition. The other 10 types can be possibly caused by race condition.

We further investigated 28 specific bugs that have all the 10 types included and confirmed 8 out of 28 bugs are caused by race conditions. 20 out of the 28 bugs might be caused by race condition, but further investigation is needed. This preliminary study gives us confidence that race condition bugs in syzbot are not rare. Considering the total amount of open bugs in syzbot, there can be a large amount of race condition bugs that do not have reproducers. For a race condition bug that does not have a reproducer, it is very hard for an analyst to debug and find the root cause of it. During the process of manually investigating the 8 race condition bugs that do not have reproducers, we learned that for someone that does not have a deep understanding of the kernel, just relying on the bug report is very hard to understand the bug's behavior and root cause. But once we figured out a thread interleaving that triggers the bug, it becomes very intuitive to figure out how to fix a race condition. So our motivation behind Macaron is that we believe that providing reproducer that can deterministically and reliably reproduce a race condition bug, which can greatly help the analyst understand and then fix it.

Timestamps	Bugs with Reproducers	Bugs w/o Reprocuers	Total Bugs	Bugs w/o Reproducers / Total Bugs
12/15/2020 18:48:56 UTC	2401	735	3136	23%
08/09/2021 20:28:00 UTC	2850	956	3806	25%
08/01/2022 12:50:00 UTC	3593	1218	4811	25%

Table 3.1: *Syzbot Bug Study*

3.3 Design

Macaron's goal is to automatically and reliably reproduce kernel race condition bugs without reproducers in syzbot. To achieve this goal, there are four requirements we need to satisfy in Macaron's design: 1. Being automatic 2. Being reliable 3. Being efficient 4. Requiring No modifications to syzbot. In this section, we explain the challenges and solutions to satisfy

those four requirements.

Requirement 1. Automatic. This means the whole reproducing process should be taken care of by Macaron without requiring the analyst's expertise on the kernel and human intervention during the reproducing process. One does not need to understand the race condition bug or know anything about the kernel to use Macaron. Some kernel-specific semantics should be incorporated into Macaron's design to optimize the searching process. We make the reproducing process automatic and easy to use for whoever wants to reproduce the bug.

Requirement 2. Reliable. This means that once Macaron reports a reproducer, it must be a true positive. To satisfy this requirement, correctly initializing the kernel and reproducing the bug in a reliable dynamic environment is the first step. Because we do not want to introduce false positive, we naturally forsake static analysis as our key technique but instead using it as a complimentary technique. We will explain the use of static analysis in section §3.5. To run a kernel in a reliable environment, we can either directly use virtual machine such as QEMU [24] or use S2E [44], a dynamic platform that implemented selective symbolic execution. We chose S2E as our underlying environment because it not only can guarantee the kernel runs in a reliable environment but also has a lot of nice features that can help to satisfy the Requirement 3. We use the features from SSE to enables delicate program analysis and improve the overall efficiency in reproducing race condition bugs.

Requirement 3. Efficient. Our work focuses on reproducing already discovered bugs and hence differs from other kernel fuzzing works [98, 89, 65, 70, 58] that improve on finding more kernel race condition bugs. Since the bugs Macaron targets have already been confirmed to exist, we keep an important design requirement in mind that they should be able to be reproduced in a limited time. In other words, the process should be as efficient as possible. Offline information about bugs that includes stack dump, core dump and text description have been shown to be very helpful in generating exploits [124, 27]. Sharing similar idea

of utilizing offline bug related information, we can narrow Macaron's exploration space by leveraging the bug report and log from syzbot. We designed Macaron to do an efficient guided state space search. We call our reproducing process Guided Selective Symbolic Execution (GSSE). GSSE is explained in detail in section §3.4.

Requirement 4. Compatible. This means the framework should require no modification to syzbot. We do not want to modify the current design of syzbot for two reasons. 1. Modifying syzbot to record additional runtime threads scheduling information during fuzzing will definitely help in reproducing. But the heavy overhead caused by dynamic recording is usually unacceptable in kernel production fuzzing. Fuzzing itself can be a popular analysis technique mainly due to its efficiency. Any changes that add noticeable overhead will greatly impact a fuzzer's ability of finding bugs. 2. If reproducing requires modifications to the current fuzzing platform, all the old race condition bugs reported by syzbot cannot be solved. While non-reproducible race condition bugs are an important portion of the existing bugs in syzbot based our study in section §3.2.

3.4 Guided Selective Symbolic Execution

In this section, we explain our GSSE in details. GSSE is dynamic SSE guided by a distance-based interleaving points prioritizing algorithm. We first use an illustration example to show how we can guide the SSE of a multi-threaded program to find a thread scheduling that reaches a bug location quickly. Then we explain the distance-based threads prioritizing algorithm that supports GSSE.

3.4.1 Illustration Example

The example is illustrated in Figure 3.2. It has two threads, one main thread and one child thread. To trigger the assertion in the main thread at line 24, a specific thread execution sequence like *(L3, L14, L20, L4, L7, L21, L24)* needs to be satisfied. In such a multi-threaded scenario, there can be multiple possible thread interleavings. Our goal in this example is to find a thread scheduling that can trigger the assertion as soon as possible. We use the inference rules in SIFT [122] to find the interleaving points and use a distance-based interleaving points prioritizing algorithm to guide the example's SSE.

Now let us go through this example to see how the assertion can be triggered within just 9 execution states. In this example, the first round of SSE without any forced context switching discovers 5 IPs: 3 IPs that access global variable k and 2 IPs that access global variable z, shown in Figure 3.3. To guide the next round's execution, we want to sort those 5 IPs by their distances to the target bug location at line 24. The distance of each IP is decided by querying the example's Multi-threaded Program Dependency Graph (MPDG). A program's MPDG reveals the relevance of two interleaving points. In other words, the closer the two IPs in the MPDG, the shorter their distance is and the more relevant they are to each other. By sorting all IPs' distances to the target location, we know how relevant all the IPs' are to the target location. Such MPDG is generated using static analysis and augumented by dynamic SSE. We will further explain MPDG in section §3.5 and how to generate it. Now let us assume we already have the program's MPDG. In the MPDG, we find that IP1, IP2 and IP3 have no available path to the target location at line 24. The distance for IP4 is 20 and for IP5 is 21. So out of all 5 IPs, we choose IP4 with the shortest distance in the main thread into a *chosenIP* set shown in Figure 3.4 for the use of next round's SSE. Every time a new IP is added into a *chosenIP* set, it will be assigned a $weight = ++GlobalHighestWeight$. *GlobalHighestWeight* is used to keep track the global highest weight for all the IPs and its initial value is 0. And all the weights of the existing IPs in the *chosenIP* set will be plus one.

The *chosenIP* set contains all IPs chosen to be considered as potential race locations by the next round's execution. In this example, after the first round's SSE, now the *chosenIP* set only has one IP which is IP4 with *weight = 1*. The second round of SSE executes IP4 first then switches to the child thread. But this round of SSE discovers no new IPs. Based on the feedback from the second round, we added IP5 that has the next shortest distance to *chosenIP* set. Now the *chosenIP* set has two IPs shown in Figure 3.5: IP4 with *weight = 1* and IP5 with *weight = 2*. *GlobalHighestWeight* now equals to 2.

As it continues, the third round that can trigger the assertion with a set of chosen IPs is shown in Figure 3.6. At the end, we terminate the whole exploration and extract the threads scheduling that triggers the assertion. The total number of states being executed to trigger the assertion using our distance-based interleaving points prioritizing algorithm is 9. As a comparison, SIFT executes over 1000 states to trigger this assertion.

3.4.2 Distance-based Interleaving Points Prioritizing Algorithm

From the illustration example, we can see that GSSE is very efficient in reaching a target location. The intuition behind the distance-based interleaving points prioritizing algorithm is that if a thread interleaving brings us closer to the target location, we should prioritize it. The shortest distance among all the newly discovered IPs is a metric we use to measure if we are getting closer to the target location. In other words, the smaller of all the newly discovered IPs' shortest distance, the closer we are to the target location and it is more likely to trigger a bug. If there is no newly discovered IPs after one round of execution, we keep adding new IPs to the *chosenIP* set and also keep the old IPs already in it. We now explain distance-based interleaving points prioritizing algorithm in details with the pseudo code in Figure 3.7 and Figure 3.8 .

We first explain how we update the *chosenIP* set and keeps the main execution loop of GSSE

```
1  int x, y, w, z, k;
2  void *accessorThread(void *arg){
3     z ++;
4     if(k > 0)
5     k = 0;
6     if(w > 0)
7     y = 0;
8  }
9  int main(int argc, char *argv[]){
10    int res = 0;
11    int *l_y;
12    pthread_t acc;
13    pthread_create(&acc,NULL,accessorThread,NULL);
14    if(z > 0)
15        w++;
16    k = 1;
17    l_y = &y;
18    *l_y = 1;
19    x = 1;
20    y = 2;
21    if(y == 0)
22        x--;
23    if(x <= 0)
24        assert(0);
25    pthread_join(acc,(void**)&res);
26    return 0;
27 }
```

Figure 3.2: *Multi-threaded GSSE Illustration Example*

going. The algorithm is shown in Figure 3.7. N controls the number of rounds of the whole execution, defined by the analyst. Since exploring all the interleaving points of the kernel is impossible in practice, we need a termination strategy for GSSE. GSSE's termination is determined either by the number of rounds of GSSE specified by the analyst or when the bug is successfully triggered. A round of GSSE stops when all the threads interleavings have been explored based on the IPs in the current *chosenIP* set. The logic of state exploration based on the *chosenIP* set is hidden in the function *exploreIPsByWeight*, explained next in Figure 3.8.

Specifically, in each round of GSSE, it generates a set of interleaving points from all the state executions in the current round. We will do post-processing for those generated IPs to update our global *sortedIP* set and *chosenIP* set. *sortedIP* has all the IPs sorted based

```
IP1  %1 = load i32, i32* k (L4)
IP2  store i32 0, i32* k (L5)
IP3  store i32 1, i32* k (L16)
IP4  %0 = load i32, i32* z (L14)
IP5  store i32 %inc, i32* z (L3)
```

Figure 3.3: *First round discovered IPs of the illustration example*

```
IP4  %0 = load i32, i32* z (L14) with weight = 1
```

Figure 3.4: *Second round chosenIP set of the illustration example*

on their distances to the target IP in the MPDG. *chosenIP* contains all the IPs that will be considered as the scheduling points in the next round's execution. We update each round's of *chosenIP* set based the results of the previous round. There are three different cases that we update the *chosenIP* set, shown from line 6 to line 14. If a new IP with a shorter distance to the target location is discovered, we add this new IP to our *chosenIP* set and update the weights of all the existing IPs in the *chosenIP* set. If no new IP with a shorter distance is discovered, we add another IP that has the next shortest distance to the *chosenIP* set. If the shortest distance from the sortedIP is smaller than its old value, we think the current thread scheduling is bad to keep. So we swap one IP in the *chosenIP* set with a different IP.

Now we explain how we make threads switching decisions in each round based on all the IPs in the *chosenIP* set, which is the logic hidden in function *exploreIPsByWeight* in Figure 3.8. During the execution, when an IP of a thread in the *chosenIP* set is hit at line 1, we will compare this IP's weight with the weights of all other threads' next IP. Based on the comparison result, there are three possibilities in terms of the continuing execution: 1. If the current thread has the maximum weight of all the IPs, we continue executing the current thread, shown at line 8. 2. Fork new execution states if there are some other threads have the same highest weighted IP with the current thread. The forked states will schedule new threads with the highest weight. The current state will continue its execution without thread switching, shown from line 10 to line 14. 3. Fork new execution states if there are some other threads have the highest weighted IP. The forked states will schedule new threads with

```
IP4  %0 = load i32, i32* z (L14) with weight = 2
IP5  store i32 %inc, i32* z (L3) with weight = 1
```

Figure 3.5: *Third round chosenIP set of the illustration example*

```
IP5  store i32 %inc, i32* z (L3) with weight = 4
IP4  %0 = load i32, i32* z (L14) with weight = 3
IP6  store i32 0, i32* y (L7) with weight = 1
IP7  store i32 2, i32* y (L20) with weight = 2
```

Figure 3.6: *Final round chosenIP set of the illustration example*

the highest weight. The current state will stop its execution, shown from line 16 to line 18.

We believe that to reach a specific location, the interleaving points should be treated differently. GSSE greatly improved the efficiency of exploring all the discovered IPs. By giving each IP a different weight, SSE is guided with a specific goal of reaching the target location as soon as possible.

N : number of rounds defined by the analyst

1: sortedIP ← {ϕ}
2: chosenIP ←{ϕ}
3: oldSortedIP ←{ϕ}
4: for i : 1 to N do
5: {sortedIP, new states} ← exploreIPsByWeight(current_state, chosenIP)
6: diff ← getShortestIPDis(sortedIP) - getShortestIPDis(oldSortedIP)
7: if diff > 0
8: chosenIP←replaceOneIP(oldSortedIP)
9: else if diff < 0
10: chosenIP←addOneIP(sortedIP)
11: oldSortedIP ← sortedIP
12: clear sortedIP
13: else
14: chosenIP←addOneIP(oldSortedIP)
15: end if
16: end for

Figure 3.7: *Algorithm 1: The main algorithm to update chosenIP set*

```
inputs {cur: current state; cur.thead: current thread that is executing}
outputs {succ: all the new forked states}

1: exploreIPsByWeight(cur, chosenIP)
2: if cur.thread.pc ∈  chosenIP is true then
3:      let threads ths ←  threadsWithMaxWeight(cur)
4: if number of ths is 0 then
5:       do not schedule any other thread and keep executing the current one
6: end if
7: if weight of cur.thread.pc  > weight of IP in ths[0]
8:      do not schedule any other thread and keep executing the current one
9: else
10:     for each thread t' in ths
11:             if is enabled(s, t') then
12:                     new state s'  ← cur.copy()
13:                     stop cur.thread, start thread t' in s'
14:                     succ ←  {s'} ∪  succ
15:     end for
16:     if weight of cur.thread.pc  < weight of IP in ths[0]
17:             terminate current state
18:     end if
19: end if
20: return succ
```

Figure 3.8: *Algorithm 2: The threads interleaving points scheduling based on their weights*

3.5 Multi-threaded Program Dependency Graph

MPDG supports the distance-based interleaving points prioritizing algorithm. As we mentioned in section §3.4.2, we use MPDG to query the distance of each IP. It serves the purpose of revealing the relevance of an interleaving point to the target location. Program Dependency Graph (PDG) is well known as a graph that makes both the data and control dependencies for each operation in a program explicit. Data dependencies are used to represent the relevant data flow relationships of a program. Control dependencies are introduced

to represent the control flow relationships of a program. We introduce MPDG to reveal the dependencies of each operations in a multi-threaded program. MPDG is first built from PDG using data-flow analysis and control-flow analysis. But static analysis is known to be inaccurate and can miss important points-to information, especially for the kernel. We augment the MPDG with dynamic information from GSSE. After each round of GSSE, if we see two operations access a shared memory object but there is no dependency flow between them in the current MPDG, we add a dependency edge between them to the graph. In the case that such two operations come from different threads, only GSSE can accurately detect them. The iterative augmentation from dynamic information to a statically built PDG makes our MPDG unique and more accurate.

Now we use the MPDG built for the illustration example in section §3.4.1 to explain the nodes and edges of the MPDG. In Figure 3.9, the nodes in the MPDG are all LLVM instructions. Specifically, the instruction in the green rectangle is the definition of a shared variable. The instruction in the red rectangle is a load instruction from a shared variable. The instruction in the blue rectangle is a store instruction of a shared variable. The instruction in the black rectangle is all other instructions that are relevant to a shared variable, for example, a branch instruction based on a shared variable's value. As for the edges between the nodes, there are four types of them. From the static analysis, we derive three types of edges. 1. Direct def-use edges, represented by the black lines. Those are the edges between variable definitions and their direct uses. 2. Direct def-use edges, represented by the black dotted lines. Those are edges between the same variable's uses. 3. Control-flow dependency edges, represented by the dotted blue lines. Those are the edges in a conventional inter-procedural control flow graph. There is a fourth type of edge that is augmented from GSSE. Those are the inter-thread dependencies of instructions on the same memory object, represented by the solid red lines. Each edge has a distance of 1. The distance of an IP is calculated by the number of edges in the path from the IP to the target location. When we want to query an IP's distance from the MPDG, we first match this IP to the node in the graph and then

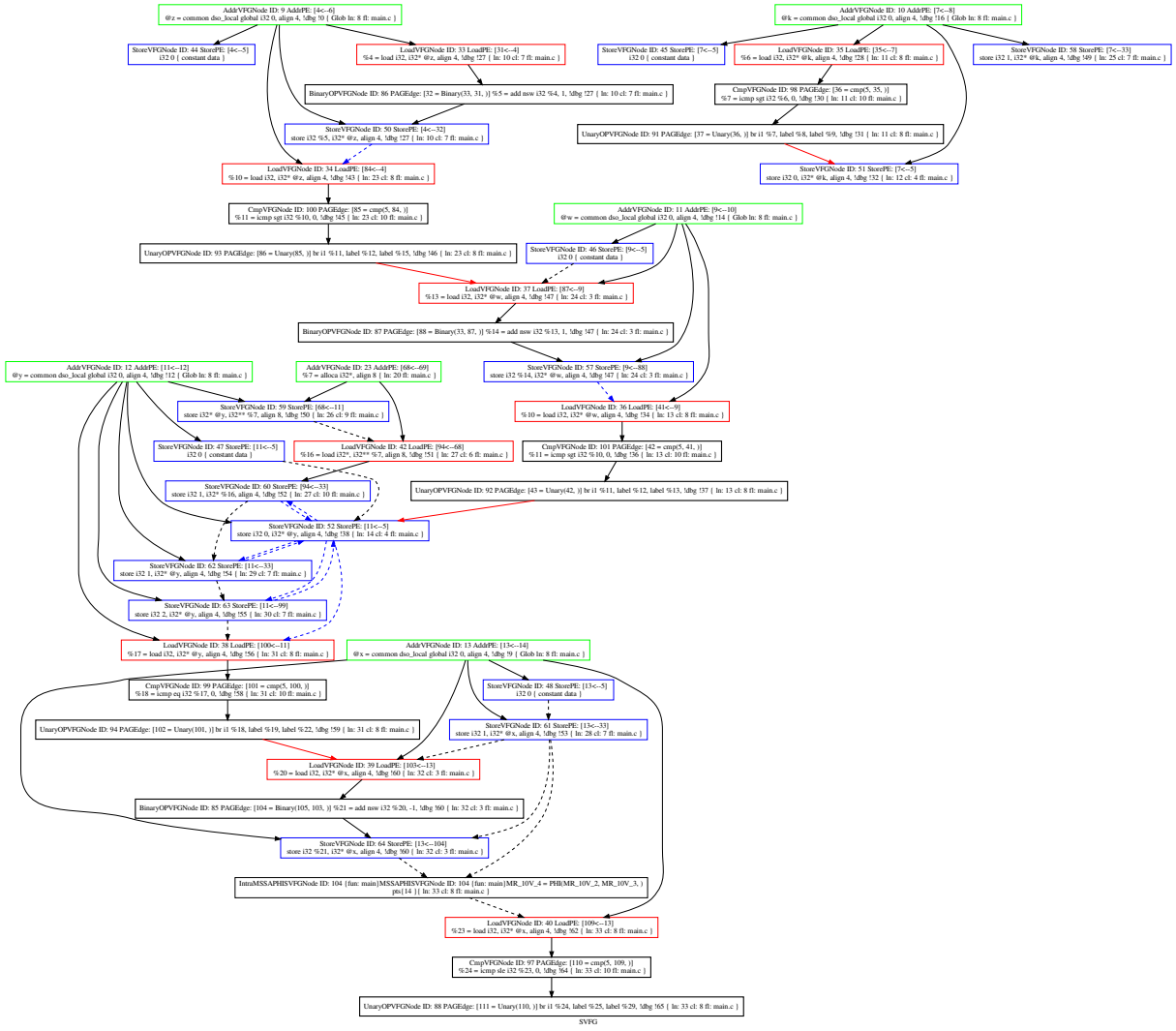


Figure 3.9: *Illustration example's MPDG.*

return the shortest distance between the matched node and the target node as the result.

3.6 Implementation

We built Macaron on top of S2E. S2E is composed of three components: QEMU, KLEE and the core S2E libraries. We heavily modified KLEE in S2E to integrate with SIFT's threads interleaving points inference rules and a static value-flow analysis tool called SVF [106]. Our

distance-based threads interleaving points prioritizing algorithm is implemented in the core S2E libraries. We also added a kernel thread creation monitoring plugin, a global memory tracing plugin and an interleaving points post-processing plugin to S2E. In this section, we explain some of the implementation challenges we faced when implementing Macaron according to its workflow shown in Figure 3.10.

3.6.1 The Test Program

As we can see in the Figure 3.10, The first step to reproduce a race condition bug is to find a potential test program before doing any thread scheduling in GSSE. The test program selection process is unique to bugs in syzbot. For bugs without reproducers in syzbot, it will provide a large log file that has all the test programs that have been run up to the bug is triggered. A bug report is also provided. To find the potential test program out of a large pool, we leverage the stack dump of a bug reported in syzbot. We reuse the reproducing process of syzbot. But as each test program is executing, we also use kprobe in the kernel to match a function location in the bug's stack dump. A matched location means all the call stacks before this location are matched exactly. The test program that matches the highest function location in the stack dump will be selected as a potential test program. It is noteworthy that this step is not sound, which means the selected potential test program may not be the exactly the one we need to trigger a race condition bug. But in most cases, we are able filter out all of the test programs that are irrelevant to the bug and find one potential test program.

3.6.2 Kernel Instrumentation

In the next step, before we feed a kernel that contains a race condition bug to GSSE, we need to do some instrumentation to the kernel. There are three parts of instrumentation

we need to do: 1. Add a S2E module to the kernel for use with the S2E analysis. This is easy to do by following the instructions from S2E. 2. Add a kprobe module to the kernel for stack matching of the test program. Macaron provides a python program that can automatically patch a given kernel and its bug report with a kprobe module specifically for each bug. 3. Add customized S2E instructions to enable the tracing of all the global memories and thread creation events in the kernel. This part only requires a few lines of code to be added to the kernel's specific files. We intend to make this automatic in the future. 4. Add GOTO macro after each line of the kernel code that we want to enforce threads scheduling. We use the method from bowknots [109] to automatically do this part. Even though Macaron requires some instrumentation to the kernel, the whole process is very easy and straightforward and can support different versions of the kernel source code. After doing the kernel instrumentation mentioned above and get a specific test program, we can run a kernel in Macaron to let it reproduce a race condition bug in GSSE. Before starting GSSE, there are two configurations we need to specify in Macaron: 1. total maximum number of testing rounds, 2. the bug location in the kernel.

3.6.3 IP Recording Optimization

Recording dynamic interleaving points in the kernel is very important in GSSE. As we mentioned before, we borrow the IP recording idea from SIFT to decide how to record the interleaving points. However, for the kernel, blindly recording all the interleaving points will be impractical. In our initial try, we observed an IP explosion problem, which means there are too many IPs that we will record just using SIFT without any optimizations. IP explosion eventually causes severe problems since the recording process will never end and the post-processing of the recorded IPs will not be possible. To avoid IP explosion, we need to constraint the recording scope of IPs. We did three optimizations to solve IP explosion problem. First, we initialize the kernel in the concrete mode of S2E and do not

start recording any IP during the kernel's initialization. Second, we monitor the execution of all the bug-related threads. We use stack matching and a kernel threads creation plugin added in GSSE to monitor the progress of all the bug-related threads. When they have been executed to match a certain number of function calls in the stack, we then switch the whole execution to symbolic mode. IP recording is only enforced for the kernel code running in symbolic mode. The number of callers are tunable for different bugs. Finally, we allow symbolic mode and concrete mode switching even after IP recording starts. All the irrelevant modules that are not important for the bug will still be executed in concrete mode without any IP recording. The bug relevant modules will be executed in symbolic execution with IP recording enabled. Those three optimization greatly reduced the size of recorded IPs and solved the IP explosion problem.

3.6.4 Threads Controlling

Another important part in GSSE is to do kernel threads controlling. Whenever the GSSE wants to do a specific threads scheduling, we need to let the kernel's internal scheduler know such scheduling decision from GSSE. To achieve this, we instrument the kernel to add a GOTO macro after each source line we want to control. The GOTO macro is a while statement that checks on the value of the flag field in the current thread's task struct. We added a new field called flag in the kernel's task struct implementation. If the flag in the current thread's task struct is set to 1, the current thread will go to sleep for a few milliseconds (set to 5 for all experiments). If the flag is unset to 0, the current thread will continue without sleeping. By setting or unsetting the flag value in each thread's task struct, we achieve the goal of kernel thread controlling implicitly. The flag's value is controlled in GSSE. We pass the virtual address of the flag field in each thread's task struct when the thread is created. GSSE can then write the value of a flag to its virtual address directly. We use a single processor in the virtual machine to make the thread controlling easier in

our implementation. We realized that adding GOTO macro in the kernel's source code can require heavy instrumentation to the kernel even if it is automatic as mentioned in 3.6.2. So we plan to use kprobe to eliminate this part of instrumentation in the future. We can add kprobe handlers at different locations of the kernel and each handler function contains the GOTO macro.

3.6.5 MPDG for Kernel Modules

The last step in Macaron's workflow is to generate a MPDG for kernel modules. To generate a MPDG for a kernel module, we use SVF [106] as our static analysis tool. We added it as a library to S2E. SVF accepts a LLVM bitcode as input and does value-flow and control-flow analysis. In Macaron, the input bitcode is compiled and linked with all the bug-related kernel modules. The exact modules we compile into one bitcode are decided by all the kernel files appeared in the bug's stack dump. Since the kernel is such a big software, compiling all of its source code into a bitcode for static analysis will be too expensive. Besides, compiling the whole kernel to LLVM is not even necessary for a specific bug. The SVF library runs analysis on this bitcode file and generates an initial MPDG before the dynamic execution starts. We store the global distance information of all the instructions in the MPDG into a hashmap to speed up the query later. The distance is defined as the smallest number of edges from a specific instruction to the target instruction. The target instruction is specified by the analyst as the bug location. The MPDG will be augmented later by the dynamic IP information from GSSE as mentioned in §3.5.

3.6.6 IP Matching

Once a dynamic IP set is recorded after Macaron finishes its current run, we sort all the recorded IP based on the kernel's MPDG. To get the distance of each IP in the dynamic IP

set, we need to match each dynamic IP from S2E to its LLVM Instruction in the MPDG. However, there is a problem in directly matching these two different versions of instructions. The LLVM instruction executed in S2E is lifted from the kernel's binary code while the LLVM instruction in the MPDG is directly compiled using clang from the kernel's source code. They have different LLVM representations for the same binary code. To match the same instruction from S2E to its representation in the MPDG, we generate the source line information for LLVM instructions both in S2E and in the MPDG. We then use the source line information of two representations of the LLVM instructions as a common ground to do the matching. In other words, If the debug information of a dynamic IP matches the debug information of an IP in the program dependency graph, we can say the two IPs are matched. The distance for this dynamic IP is the distance for its matched IP in MPDG.

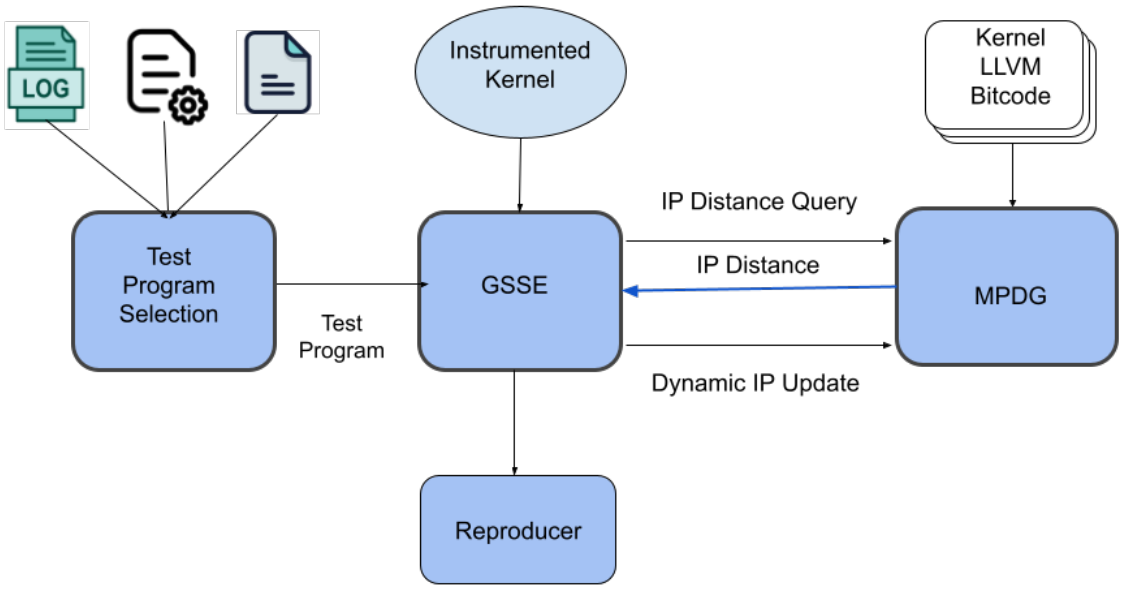


Figure 3.10: *Macaron workflow*

3.7 Evaluation

In this section, we evaluate Macaron's performance on reproducing different kernel race condition bugs.

3.7.1 Bug Case Study

Before we try to use Macaron to automatically reproduce some race condition bugs, we spent a great effort to manually study 8 use-after-free bugs and created reproducers with proper thread controlling by adding semaphores in the kernel to trigger those bugs 3.2. According to our study, 4 out of the 8 bugs have one race condition, which requires two context switches to be triggered. 1 out 8 cannot be manually reproduced. 3 out 8 have two race conditions, which require four context switches. Macaron now can only deal with race condition bugs with one race condition and cannot deal with race condition bugs that have more than one race condition. At the time this thesis was written, we successfully reproduced the first bug listed in table 3.2. We show the experimental results of reproducing this use-after-free bug using Macaron. Our goal is to see how well Macaron performs to find a reproducer.

3.7.2 Experiments

The use-after-free that found in the `rc_dev_uevent` function happens when a process unregisters a USB gadget device. The test program first registers a USB device and allocates different memories for it. When the user space test program exits, it unregisters the USB gadget device. The user space test program then delegates a kernel thread to send a uevent notification to an OS daemon called `udev` which manages the system's device events. The `udev` daemon processes the uevent notification and read the content in the device's `rc_map` memory. But at the same time, a kernel thread has freed the registered device's `rc_map` mem-

ory. This bug was captured by the Kernel Address Sanitizer (KASAN) [10] as a use-after-free bug.

We conducted three experiments for this use-after-free bug in Macaron by changing three variables: 1. The IP trace starting point (which is decided by the execution stage of the bug-related threads). 2. The size of kernel bitcode compiled from the source files to generate the MPDG. 3. Whether to use PDG that only uses static analysis or MPDG that is augmented by dynamic GSSE on PDG. We keep the other two variables fixed while changing one of the variables. In table 3.3, we change the IP trace starting point. N/A in the table means we cannot reproduce the bug. The late stage of threads' execution means we will start recording IPs when the bug's use and free threads reach their latest callers in their call stack, accordingly, early and middle. This experiment shows that starting the IP tracing from late stage of the threads' execution helps to reproduce a bug. While starting the IP tracing from early or middle stage of the threads' execution cannot reproduce the bug because that will cause Macaron to record irrelevant threads and Macaron can only handle two threads' interleaving. We use MPDG and use a small size of kernel bitcode in this experiment. Experiment shown in table 3.4 shows that the size of the kernel bitcode does not matter much. We use MPDG and start the IP tracing from the late stage of the threads' execution. The large kernel bitcode means we compile all the files shown in the bug's stack dump. The small kernel bitcode means we only compile the last few files in the bug's stack dump. The takeaway from this experiment is that we should always start with a smaller kernel bitcode to improve the performance. The last experiment in table 3.5 shows that MPDG helps to reproduce a bug. While with just using PDG, the bug cannot be reproduced. This experiment shows that the augumented information from GSSE is very important.

Bug	# of Race conditions	Manually reproducible
KASAN: use-after-free Read in rc_dev_uevent	1	Yes
KASAN: use-after-free Read in ovl_real_fdget_meta	1	Yes
KASAN: use-after-free Read in uprobe_munmap	1	Yes
KASAN: use-after-free Write in addr_resolve (2)	1	Yes
KASAN: use-after-free Read in vlan_dev_real_dev	N/A	No
KASAN: use-after-free Read in skb_release_head_state	2	Yes
KASAN: use-after-free Read in hci_dev_do_open	2	Yes
KASAN: use-after-free Read in recv_work	2	Yes

Table 3.2: *Bug Case Study*

IP trace starting point	Total num of states to reproduce	Total generated IPs	Time (mins) to reproduce
Late stage of threads' execution	4	374	24
Middle stage of threads' execution	N/A	N/A	N/A
Early stage of threads' execution	N/A	N/A	N/A

Table 3.3: *Different IP trace starting point evaluation*

3.8 Discussions

3.8.1 Reproducer

A reproducer of a race condition bug in Macaron includes a test program and a specific thread scheduling in the kernel. The test program is very intuitive; it is whatever the input program we feed to the kernel to reproduce a bug. While the thread scheduling can be represented in two different ways. First, when Macaron reproduces a bug, it will report the exact interleaving points and their orders in different threads that we do context switching. Macaron naturally understands such kind of thread scheduling output by itself. If the analyst

Kernel bitcode size	Total num of states to reproduce	Total generated IPs	Time (mins) to reproduce
Small kernel bitcode	4	374	24
Large kernel bitcode	4	403	27

Table 3.4: *Different kernel bitcode size evaluation*

Graph type	Total num of states to reproduce	Total generated IPs	Time (mins) to reproduce
PDG	N/A	N/A	N/A
MPDG	4	374	24

Table 3.5: *PDG vs MPDG evaluation*

is willing to set up Macaron following the instructions and enforce such thread scheduling, they can easily reproduce the bug. Second, if the analyst does not want to use Macaron but instead would like to run the kernel natively with the thread scheduling enforced directly to the kernel, they can instrument the kernel with corresponding semaphores to reproduce the bug. We provide an example reproducer 3.11 of the first bug listed in table 3.2 in this way. Furthermore, Macaron can possibly provide an automatic patch containing the semaphores that can be applied directly to the kernel.

3.8.2 Limitation

Macaron has its limitation in reproducing complex race condition bugs that have more than one race condition. Each race condition require two forced context switches. Race condition bugs that require more than four context switches are generally hard to control and sometime involve controlling more than two threads. Macaron now only traces two threads and control their interleaving. But improving Macaron to reproduce complex race condition bugs is possible. We now lack the implementation on that aspect and more experiments to understand the complexity to reproduce them. We leave this for our future work.

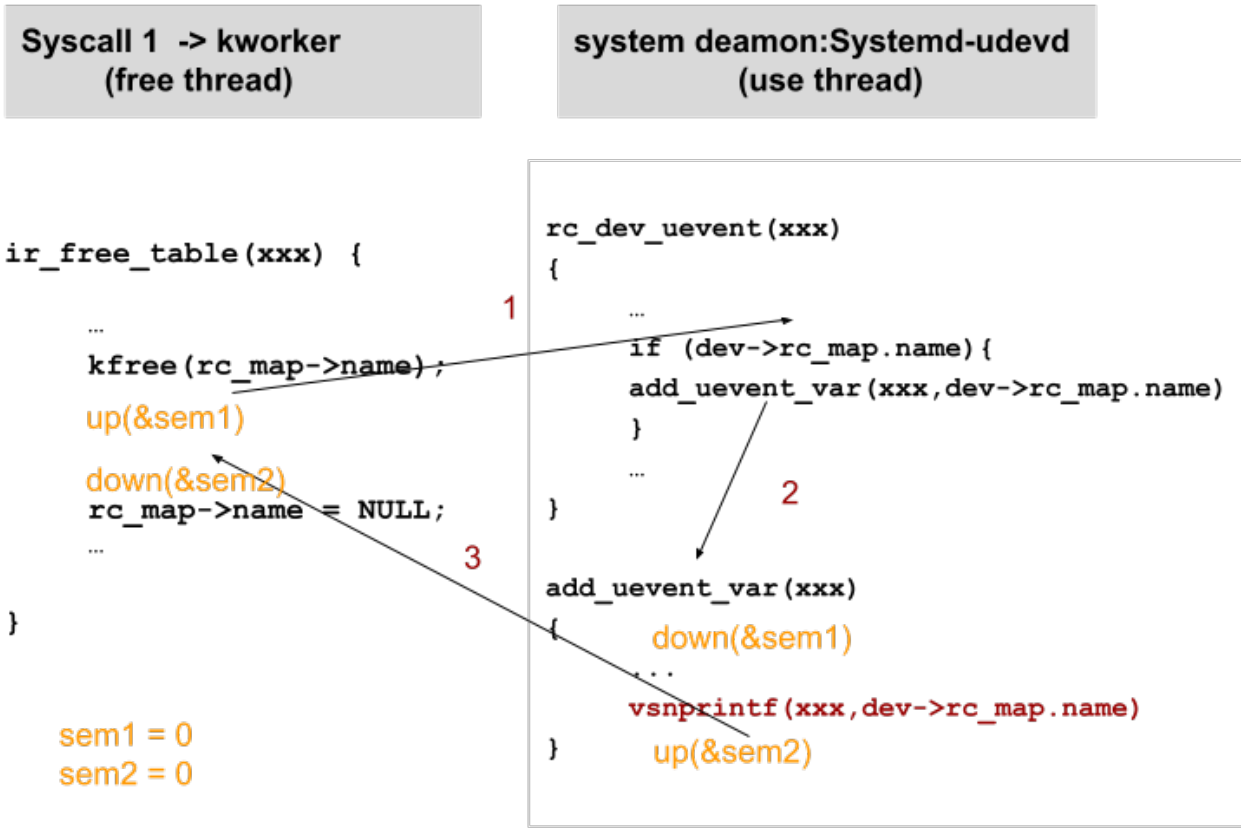


Figure 3.11: *Example reproducer with semaphores*

Chapter 4

Related works

4.1 Symbolic and Concolic Execution

Symbolic execution [22, 34, 30, 69, 118, 112] can automatically generate test inputs that execute different paths in a program. The idea of symbolic execution is instead of running code on manually constructed input, it runs the code on symbolic input. At the beginning, such symbolic input does not have any constraints and can be anything. The symbolic input may propagate to other symbolic values along with the execution of some instructions. When the program execution branches based on a symbolic value, the system forks to two paths or states. Each path maintains a set of constraints that satisfies the conditions on the symbolic values of the current path. When a path terminates or hits a bug, a test case can be generated by solving the current path constraints to generate concrete inputs. The goal of symbolic execution is to avoid generating repeated inputs that execute the same path. Instead, it automatically generate unique inputs that each one of them executes a different path. This idea shows to be very promising in exploring as many execution paths as possible in a program.

But symbolic execution also faces a lot of practical challenges for real-world programs, as mentioned in KLEE [33]. KLEE leverages several years of lessons from their previous tool [35] and employs a variety of constraint solving optimizations to make symbolic execution possible for real-world programs. The work of KLEE marks a big step in pushing symbolic execution as one of the powerful program analysis technique. A lot of works [42, 97, 48, 125, 41, 40] on symbolic execution emerged to push symbolic execution's practical applications in different areas like reverse engineering and hybrid fuzzing after KLEE. RevNIC [42] takes a closed-source binary driver, automatically reverse engineers the driver's logic, and synthesizes new device driver code that implements the exact same hardware protocol as the original driver. It introduces a technique for tracing the driver/hardware interaction and turning it into a driver state machine by using binary symbolic execution and achieves high-coverage reverse engineering of drivers. In hybrid fuzzing, symbolic execution is usually used as a form of its variant named concolic execution as explained next.

DART [57], CUTE [99] and CREST [32] combine concrete and symbolic execution to generate directed inputs, known as concolic execution. As explained in paper [36], unlike classical symbolic execution, since concolic execution maintains the entire concrete state of the program along an execution, it needs initial concrete values for its inputs. Concolic testing executes a program starting with some given or random input, gathers symbolic constraints on inputs at conditional statements along the execution, and then uses a constraint solver to infer variants of the previous inputs in order to steer the next execution of the program toward an alternative feasible execution path. Concolic execution is very useful when directing symbolic execution to a specific path and collect the path's relevant information. QSYM [125] is a fast concolic execution engine. It supports hybrid fuzzing by tightly integrating the symbolic emulation with the native execution using dynamic binary translation, making it possible to implement more fine-grained and instruction-level symbolic emulation. SAVIOR [41] is a new hybrid testing framework pioneering a bug-driven principle. It prioritizes the concolic execution of the seeds that are likely to uncover more vulnerabili-

ties. Mousse also uses concolic inputs to drive the execution in a desired path when doing distributed execution (§2.5.1) to avoid replicates.

4.2 Handle External Environment Interactions in Dynamic Testing

Mousse is the first framework specialized for programs interacting heavily with external environments in symbolic execution. During dynamic testing, either fuzzing or dynamic symbolic execution, the testing program's interactions with external environments that are outside of the testing components have to be dealt with probably. The responses from external environments directly impact the continuing execution of the testing program. If the responses from external environments are not accurate or available, there can be many false positives or the lost potential code coverage in the testing. Even worse, it can hinder the whole execution of the program to cause unexpected behaviors. There are different solutions to deal with external environment interactions in dynamic testing but none of them is suitable for programs with untamed environments as we explained in Chapter §2.

Modeling the external environments is what most of the works do when doing symbolic execution for complex programs. KLEE provides a limited system call models to handle the programs' interaction with the underlying OS. Works like AEG [19, 20] and Driller [105] that use symbolic execution to automatically generate exploits or use concolic execution to enhance the performance of fuzzing all model the environments. PROMPT [123] provides API models as one form of environment modeling and has been used to detect memory vulnerabilities in the Linux kernel. Cloud9 [47] provides a new symbolic environment model that is the first to support all major aspects of the POSIX interface, such as processes, threads, synchronization, networking, IPC, and file I/O. The insight of Cloud9 is that as few

as two primitives are sufficient to support complex OS interfaces such as POSIX: threads with synchronization, and address spaces with shared memory. Even though Cloud9 partially dealt with some of the external environment in testing the programs interacting with the OS. There is no work can model all kinds of external environments accurately in practice because that is a very stunting task. In some cases, the untamed environments developed by different vendors are all block boxes and cannot even be modeled.

To avoid modeling the external environments, Chipounov et al. proposes S2E [45] to fully virtualize the whole OS and allow transitions between symbolic execution and concrete execution. The external environments are usually running in concrete execution to avoid path explosion. S2E defines different execution consistency models when the switching happens between symbolic execution and concrete execution. Each consistency model results from different transition points between symbolic and concrete executions and hence results in a different set of program paths being analyzed.

Under-constrained symbolic execution, as mainly realized by Under-Constrained KLEE (UC-KLEE) [93, 94, 53], uses symbolic execution to analyze functions with systems code. Under-constrained symbolic execution is a lazy yet straightforward way to deal with the external environments with over estimation. It does not execute full program paths and simply considers the function arguments and returns to be symbolic. This results in lots of 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.

When fuzzing device drivers in the kernel, Charm [100] 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 that Mousse tries to deal with. 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.

Decoupled SSE is another approach to deal with the testing program's interactions with external environments. Analyzing a program with decoupled SSE requires configuring the symbolic and concrete execution engines in two separate devices and configuring a channel for memory state transfer. Mayhem [38] and Centaur [77] 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^2E, 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.

Qsym [125] 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.

4.3 Kernel Fuzzing in General

Fuzz testing history starts in the 1980s [28]. Fuzzing [81, 28, 107] provides auto-generated random inputs to programs in order to finding bugs of programs. To make the generation of inputs more efficient and intelligent, fuzzers are usually feedback-driven, especially coverage-guided. Fuzzing is very efficient in testing complex softwares. Kernel is one of the most popular targets in the fuzzing world. Kernel fuzzing has gained its popularity over the recent years. Syzkaller [2] as the state-of-the-art kernel fuzzing framework drives a lot of interesting directions for kernel fuzzing. kAFL [98] introduces a new technique that utilizes Intel's Processor Trace to provide feedback for kernel fuzzing. HFL [70] solves the challenges applying hybrid fuzzing that combines symbolic execution and fuzzing to kernel testing. MoonShine [89] is a novel strategy for distilling seeds for OS fuzzers from system call traces using light-weight static analysis for efficiently detecting dependencies across different system calls.

Because of kernel's large TCB, there are more and more fuzzers built to fuzz specific Linux kernel subsystems like device drivers [100, 103, 49] and file system [71]. DIFUZE [49] is an interface-aware fuzzing tool to automatically generate valid inputs and trigger the execution of the kernel drivers. It leverages static analysis to compose correctly-structured input in the userspace to explore kernel drivers. To fuzz file systems specifically, Hydra [71] provides building blocks for file system fuzzing, including input mutators, feedback engines, a libOS-based executor, and a bug reproducer with test case minimization.

One big challenge in fuzzing the kernel is that not only the inputs to the kernel is important but also the state of the kernel. There are some user-level stateful fuzzers [18, 23, 102] which try to consider states during fuzzing. Inspired by those works, Moonshine [90] and the work [61] explores the dependency problem in the kernel. Macaron is aware of such dependency problem in the kernel at the beginning. To trigger a race condition bug, we not only choose

a fixed inputs but also make sure the kernel is reset to the same state before the bug is triggered. Furthermore, during the dynamic execution, we explore all the different states set by thread schedulings to find a specific thread scheduling.

4.4 Kernel Race Condition Bugs Detection

Finding race condition bugs is known to be difficult with the need of exploring a large number of thread interleavings. Kernel as one of the most complex concurrent system, is implemented with fine-grained concurrency mechanisms in mind to fully utilize the power of multi-core hardware. While that concurrency optimization in the kernel greatly speeds up the whole system's execution, it is notoriously prone to race condition bugs.

Some works [52, 91, 113, 108, 111, 55, 74] use static analysis and dynamic monitoring to detect race condition bugs in the kernel. RacerX [52] is an effective tool to statically detect race conditions and deadlocks using flow-sensitive, interprocedural analysis. Relay [113] uses a static and scalable race detection analysis in which unsoundness is modularized to a few sources like ignoring reads and writes that occur inside blocks of assembly code. Veeraraghavan [111] proposes to protect a program from data race errors at runtime by executing multiple replicas of the program with complementary thread schedules. TSVD [74] uses lightweight monitoring of the calling behaviors of thread-unsafe methods, not any synchronization operations, to dynamically identify bug suspects.

There are also a lot of research works [65, 120, 39, 67] focus on fuzzing kernel race condition bugs specifically. Razzer [65] has shed lights on data race detection by combining fuzzing and static analysis. Krace [120] improves on fuzzing data races for kernel file systems with a novel notion of alias coverage and an algorithm for evolving and merging multi-threaded syscall sequences. MUZZ [39] is a new grey-box fuzzing technique that hunts for bugs in multithreaded

programs by using three novel thread-aware instrumentation, namely coverage-oriented instrumentation, thread-context instrumentation, and schedule-intervention instrumentation. CONZZER [67] is a context-sensitive and directional concurrency fuzzing approach for thread interleavings exploration.

However, fuzzing introduces a lot randomness which can be a double-edged sword for kernel race condition bugs detection. The randomness introduced by fuzzing sometimes can trigger very deep and rare race condition bugs but also miss a lot of other opportunities because of the lack of a systematic exploration of thread interleavings. Some research works [83, 31, 58, 56, 65] focus on systematic exploration of thread interleavings to improve the efficiency of searching a rare thread interleaving that causes a bug. Probabilistic Concurrency Testing (PCT) [31] is a randomized algorithm that increases the probability of funding concurrency bugs. It assumes a given input is provided and provides efficient mathematical probability of finding a concurrency bug. PPCT [84] proposes a parallel version of the PCT algorithm. SKI [56] is the first tool that fills the gap of controlling precisely which kernel interleavings are executed to find kernel concurrency bugs. It tries to expose kernel concurrency bugs through systematic schedule exploration and uses an extension of the PCT algorithm [31]. To gain the controllability of each thread's execution, It uses some heuristic to infer the liveness of threads and pin each thread to a virtual CPU. Snowboard [58] targets on kernel concurrency bugs including data races by intelligently exploring thread interleavings and test inputs jointly. It introduces a new idea called potential memory communication (PMC) to idenfity potential race locations. All those kernel fuzzing works specifically focus on concurrency bugs by improving on finding better test inputs or identifying more likely race locations to reduce search space. Razzer [65] targets on data race in the kernel using guiding fuzz testing. To reduce exploration space of data races, it identifies over-approximated potential data race spots using static analysis and prioritizes searches over those potential data race spots.

Macaron differs from all of those aforementioned works in terms of exploring the search

space of thread interleavings. Macaron uses a distance-based interleaving points prioritizing algorithm to do GSSE, which is much more efficient and goal oriented. Macaron is orthogonal to those works and can be used as a complementary reproducing tool for any kernel fuzzers built on top syzkaller.

4.5 Concurrency Bugs Reproducing and Debugging

Macaron is a work specifically for reproducing kernel race condition bugs and is hence highly related to concurrency bugs reproducing in user-space multithreaded programs. For user-space concurrent programs, the software community has done intensive research works to debug and reproduce race condition bugs.

Record-and-replay is one of the most effective ways for combating concurrency bugs. The record and replay technique [16, 72, 104, 46, 51, 85, 50] aims to fully record the problematic execution of concurrent programs. The programmers later can use the recorded information to replay the bug. However, recording such information comes with a high cost. As mentioned in LEAP [62] and CLAP [64], conventional deterministic multi-processor replay techniques usually incur a significant runtime overhead of 10x to 100x [46, 50], making them unattractive for production use or even for testing. LEAP [62] is a deterministic record and replay technique that uses a new type of local order w.r.t. the shared memory locations and concurrent threads. It records much less information without losing the replay determinism compare to other works. CLAP [64] loges purely local execution of each thread, which is substantially cheaper than logging memory interactions. Symbiosis [78] and Cortex [79] extend CLAP by isolating the root cause of the failure [78] and by reproducing failures that are control-flow dependent [79]. Despite the efforts made by many works [62, 63, 128, 64, 66] to reduce the overhead of record and replay, it is still very challenging to use this technique in the kernel race condition bugs because of both the performance reduction and the difficulty

to record the needed synchronization information.

Compared to record-and-replay, a less expensive way to reproduce race condition bugs is using the limited offline information like core dumps, crash stacks or logs to reveal the details of the bug. RECORE [96] is the first technique to use evolutionary search-based test generation to reconstruct failures from saved core dumps. D Weeratunge's work [115] analyzes multicore dumps to facilitate concurrency bug reproduction. It automatically identifies the execution point in the re-execution that corresponds to the failure point. It does so by analyzing the failure core dump and leveraging a technique called execution indexing that identifies a related point in the re-execution. ConCrash [27] utilizes the crash stacks of a bug to automatically generate test codes that reproduce concurrency failures that violate thread-safety. Crash stacks are generally easier to get compare to core dumps. Semfuzz [124] uses on-code textual bug descriptions, e.g., CVE, Linux git logs for reconstructing exploits on known vulnerabilities. It is designed and implemented as a semantics-based approach for automatic generation of proof-of-concept exploits. It works for bugs with rich textual bug descriptions and only 2.6% of the total bugs addressed in the paper are race conditions.

4.6 Other Related Work

AFix [68] is a tool that automates the whole process of fixing one common type of concurrency bug: single-variable atomicity violations. AFix starts from the bug reports of existing bug-detection tools like CTrigger [91]. It augments these with static analysis to construct a suitable patch for each bug report. However, it requires a bug report that manifests the interleaving scheduling of the bug. Macaron is different with AFix in finding such an interleaving scheduling that triggers a concurrency bug. The automation of generating a patch is also something Macaron interest to do and we leave that for the future work.

Execution synthesis [127] uses a combination of static analysis and symbolic execution, it "synthesizes" a thread schedule and various required program inputs that cause the bug to manifest. The synthesized execution can be played back deterministically in a regular debugger, like gdb. Macaron differs from [127] that it uses a different guided SSE algorithm and works for the kernel race condition bugs specifically.

Use-After-Free (UAF) bugs are usually hard-to-detect. There are works focus on detecting UAF bugs specifically during fuzzing. UAFuzz [87] is the first (binary-level) directed greybox fuzzer dedicated to UAF bugs. UAFL [114] first performs a static typestate analysis to find operation sequences potentially violating the UAF typestate property and then use fuzzing to discover vulnerabilities violating typestate properties. Its fuzzing process is guided by the operation sequences in order to progressively generate test cases triggering property violations. Macaron partially deals with UAF bug that caused by race conditions and mainly focuses on reproducing them.

Bibliography

[1] ARM TrustZone Technologies. https://www.arm.com/technologies/trustzone-for-cortex-a.

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

[3] Linux Mutex. https://docs.kernel.org/locking/mutex-design.html.

[4] Linux RCU. https://www.kernel.org/doc/Documentation/RCU/whatisRCU.txt.

[5] Linux Semaphore. https://man7.org/linux/man-pages/man0/semaphore.h.0p.html.

[6] Linux Spinlocks. https://www.kernel.org/doc/Documentation/locking/spinlocks.txt.

[7] MITRE CVE Database. https://cve.mitre.org/.

[8] Rust. https://www.rust-lang.org/.

[9] syzbot. https://syzkaller.appspot.com/upstream.

[10] The Kernel Address Sanitizer (KASAN). https://www.kernel.org/doc/html/v4.14/dev-tools/kasan.html.

[11] Trusty TEE. https://source.android.com/security/trusty/, 2017.

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

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

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

[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] G. Altekar and I. Stoica. Odr: Output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 193–206, 2009.

[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] V. Atlidakis, P. Godefroid, and M. Polishchuk. Restler: Stateful rest api fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 748–758. IEEE, 2019.

[19] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic Exploit Generation. In *Proc. Internet Society 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] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. *ACM SIGOPS Operating Systems Review*, 40(4):73–85, 2006.

[22] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *International SPIN Workshop on Model Checking of Software*, pages 102–122. Springer, 2001.

[23] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna. Snooze: toward a stateful network protocol fuzzer. In *International conference on information security*, pages 343–358. Springer, 2006.

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

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

[26] 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.

[27] F. A. Bianchi, M. Pezzè, and V. Terragni. Reproducing concurrency failures from crash stacks. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 705–716, 2017.

[28] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM systems journal*, 22(3):229–245, 1983.

[29] 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.

[30] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 15–pp. IEEE, 2006.

[31] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. *ACM SIGARCH Computer Architecture News*, 38(1):167–178, 2010.

[32] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 443–446. IEEE, 2008.

[33] 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.

[34] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. In *International SPIN Workshop on Model Checking of Software*, pages 2–23. Springer, 2005.

[35] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):1–38, 2008.

[36] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[37] 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.

[38] 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.

[39] H. Chen, S. Guo, Y. Xue, Y. Sui, C. Zhang, Y. Li, H. Wang, and Y. Liu. {MUZZ}: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2325–2342, 2020.

[40] Y. Chen, M. Ahmadi, B. Wang, L. Lu, et al. {MEUZZ}: Smart seed scheduling for hybrid fuzzing. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 77–92, 2020.

[41] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1580–1596. IEEE, 2020.

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

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

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

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

[46] J.-D. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 48–59, 1998.

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

[48] P. Collingbourne, C. Cadar, and P. H. Kelly. Symbolic crosschecking of floating-point and simd code. In *Proceedings of the sixth conference on Computer systems*, pages 315–328, 2011.

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

[50] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: Deterministic shared memory multiprocessing. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, pages 85–96, 2009.

[51] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 121–130, 2008.

[52] D. Engler and K. Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. *ACM SIGOPS operating systems review*, 37(5):237–252, 2003.

[53] 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.

[54] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. *ACM SIGOPS Operating Systems Review*, 29(5):251–266, 1995.

[55] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective {Data-Race} detection for the kernel. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[56] P. Fonseca, R. Rodrigues, and B. B. Brandenburg. {SKI}: Exposing kernel concurrency bugs through systematic schedule exploration. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 415–431, 2014.

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

[58] S. Gong, D. Altinbüken, P. Fonseca, and P. Maniatis. Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 66–83, 2021.

[59] 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.

[60] 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. Internet Society NDSS*, 2015.

[61] Y. Hao, H. Zhang, G. Li, X. Du, Z. Qian, and A. A. Sani. Demystifying the dependency challenge in kernel fuzzing. 2022.

[62] J. Huang, P. Liu, and C. Zhang. Leap: Lightweight deterministic multi-processor replay of concurrent java programs. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 207–216, 2010.

[63] J. Huang and C. Zhang. Lean: Simplifying concurrency bug reproduction via replay-supported execution reduction. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 451–466, 2012.

[64] J. Huang, C. Zhang, and J. Dolby. Clap: Recording local executions to reproduce concurrency failures. *Acm Sigplan Notices*, 48(6):141–152, 2013.

[65] 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 (SP)*, pages 754–768. IEEE, 2019.

[66] Y. Jiang, T. Gu, C. Xu, X. Ma, and J. Lu. Care: Cache guided deterministic replay for concurrent java programs. In *Proceedings of the 36th International Conference on Software Engineering*, pages 457–467, 2014.

[67] Z.-M. Jiang, J.-J. Bai, K. Lu, and S.-M. Hu. Context-sensitive and directional concurrency fuzzing for data-race detection. 2022.

[68] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 389–400, 2011.

[69] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568. Springer, 2003.

[70] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee. Hfl: Hybrid fuzzing on the linux kernel. In *NDSS*, 2020.

[71] S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, and T. Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 147–161, 2019.

[72] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the 2005 USENIX Technical Conference*, pages 1–15, 2005.

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

[74] G. Li, S. Lu, M. Musuvathi, S. Nath, and R. Padhye. Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 162–180, 2019.

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

[76] 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.

[77] 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.

[78] N. Machado, B. Lucia, and L. Rodrigues. Concurrency debugging with differential schedule projections. *ACM SIGPLAN Notices*, 50(6):586–595, 2015.

[79] N. Machado, B. Lucia, and L. Rodrigues. Production-guided concurrency debugging. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 2016.

[80] 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.

[81] B. P. Miller, G. Cooksey, and F. Moore. An empirical study of the robustness of macos applications using random testing. In *Proceedings of the 1st international workshop on Random testing*, pages 46–54, 2006.

[82] 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.

[83] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, volume 8, 2008.

[84] S. Nagarakatte, S. Burckhardt, M. M. Martin, and M. Musuvathi. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 543–554, 2012.

[85] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 284–295. IEEE, 2005.

[86] 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 *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 41–61, 2020.

[87] M.-D. Nguyen, S. Bardin, R. Bonichon, R. Groz, and M. Lemerre. Binary-level directed fuzzing for {Use-After-Free} vulnerabilities. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 47–62, 2020.

[88] 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.

[89] 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.

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

[91] S. Park, S. Lu, and Y. Zhou. Ctrigger: exposing atomicity violation bugs from their hiding places. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, pages 25–36, 2009.

[92] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the library os from the top down. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, pages 291–304, 2011.

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

[94] 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.

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

[96] J. Rößler, A. Zeller, G. Fraser, C. Zamfir, and G. Candea. Reconstructing core dumps. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 114–123. IEEE, 2013.

[97] R. Sasnauskas, O. S. Dustmann, B. L. Kaminski, K. Wehrle, C. Weise, and S. Kowalewski. Scalable symbolic execution of distributed systems. In *2011 31st International Conference on Distributed Computing Systems*, pages 333–342. IEEE, 2011.

[98] 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.

[99] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. *ACM SIGSOFT Software Engineering Notes*, 30(5):263–272, 2005.

[100] 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.

[101] 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.

[102] C. Song, B. Yu, X. Zhou, and Q. Yang. Spfuzz: a hierarchical scheduling framework for stateful network protocol fuzzing. *IEEE Access*, 7:18490–18499, 2019.

[103] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *2019 Network and Distributed Systems Security Symposium (NDSS)*, pages 1–15. Internet Society, 2019.

[104] S. M. Srinivasan, S. Kandula, C. R. Andrews, Y. Zhou, et al. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conference, General Track*, pages 29–44. Boston, MA, USA, 2004.

[105] 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. Internet Society NDSS*, 2016.

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

[107] M. Sutton, A. Greene, and P. Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[108] B. Swain, B. Liu, P. Liu, Y. Li, A. Crump, R. Khera, and J. Huang. Openrace: An open source framework for statically detecting data races. In *2021 IEEE/ACM 5th International Workshop on Software Correctness for HPC Applications (Correctness)*, pages 25–32. IEEE, 2021.

[109] 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.

[110] M. A. Vander-Pallen, P. Addai, S. Isteefanos, and T. K. Mohd. Survey on types of cyber attacks on operating system vulnerabilities since 2018 onwards. In *2022 IEEE World AI IoT Congress (AIIoT)*, pages 01–07. IEEE, 2022.

[111] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Detecting and surviving data races using complementary schedules. In *Proceedings of the twenty-third ACM symposium on operating systems principles*, pages 369–384, 2011.

[112] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with java pathfinder. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 97–107, 2004.

[113] J. W. Voung, R. Jhala, and S. Lerner. Relay: static race detection on millions of lines of code. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 205–214, 2007.

[114] H. Wang, X. Xie, Y. Li, C. Wen, Y. Li, Y. Liu, S. Qin, H. Chen, and Y. Sui. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 999–1010. IEEE, 2020.

[115] D. Weeratunge, X. Zhang, and S. Jagannathan. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*, pages 155–166, 2010.

[116] 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.

[117] 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.

[118] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 365–381. Springer, 2005.

[119] J. Xu, D. Mu, P. Chen, X. Xing, P. Wang, and P. Liu. Credal: Towards locating a memory corruption vulnerability with your core dump. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 529–540, 2016.

[120] 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 (SP)*, pages 1643–1660. IEEE, 2020.

[121] Z. Yao, Z. Ma, Y. Liu, A. Amiri Sani, and A. Chandramowlishwaran. Sugar: Secure gpu acceleration in web browsers. *ACM SIGPLAN Notices*, 53(2):519–534, 2018.

[122] T. Yavuz. Sift: A tool for property directed symbolic execution of multithreaded software. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 433–443. IEEE, 2022.

[123] T. Yavuz and K. Y. Bai. Analyzing system software components using api model guided symbolic execution. *Automated Software Engineering*, 27(3):329–367, 2020.

[124] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2139–2154, 2017.

[125] 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.

[126] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti. Avatar: A framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *Proc. Internet Society NDSS*, 2014.

[127] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *Proceedings of the 5th European conference on Computer systems*, pages 321–334, 2010.

[128] J. Zhou, X. Xiao, and C. Zhang. Stride: Search-based deterministic replay in polynomial time via bounded linkage. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 892–902. IEEE, 2012.