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

Yingtong Liu, Hsin-Wei Hung, Ardalan Amiri Sani
University of California, Irvine
{yingtong,hsinweih,ardalan}@uci.edu

Abstract

Selective symbolic execution (SSE) is a powerful program analysis technique for exploring multiple execution paths of a program. However, it faces a challenge in analyzing programs with environments that cannot be modeled nor virtualized. Examples include OS services managing I/O devices, software frameworks for accelerators, and specialized applications. We introduce Mousse, a system for analyzing such programs using SSE. Mousse uses novel solutions to overcome the above challenge. 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.

CCS Concepts: • **Software and its engineering** → **Software verification and validation**; **Distributed systems organizing principles**; • **Theory of computation** → **Program analysis**.

Keywords: selective symbolic execution, program environment, program analysis

ACM Reference Format:
Yingtong Liu, Hsin-Wei Hung, Ardalan Amiri Sani. 2020. Mousse: A System for Selective Symbolic Execution of Programs with Untamed Environments. In *Fifteenth European Conference on Computer Systems (EuroSys '20), April 27–30, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3342195.3387556

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EuroSys '20, April 27–30, 2020, Heraklion, Greece
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6882-7/20/04…$15.00
https://doi.org/10.1145/3342195.3387556

1 Introduction

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

In the past, SSE has been used to implement various types of analysis, such as bug and vulnerability detection [17, 26, 35], performance profiling [17] and reverse-engineering of binaries [15, 17]. In addition, it can be used for taint analysis, hybrid fuzzing [38, 41], and for exploit generation and analysis [6, 7].

In this paper, 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 [6, 7, 14, 37] and virtualizing it [17]. 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 [15, 26, 35], 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 [2, 31, 42]. 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 9 hours when using a single device. To address this problem, we introduce a distributed execution approach that supports concurrent execution of the analysis on multiple identical devices, while avoiding duplicate paths.

To demonstrate the benefits of Mousse, we use it to analyze five 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 19% more L1 data cache misses for some playback configurations.

```
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 1. *Simple hypothetical program used to describe the inner workings of SSE.*

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 [3].

2 Background & Motivation

2.1 Selective Symbolic Execution

Selective symbolic execution (SSE) is a powerful program analysis technique that can analyze multiple execution paths of a program [2, 14, 16–18, 29, 31, 42]. 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 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 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 §4. Both paths now

RIGHTSLINK

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 §4 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.

2.2 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 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 [6, 7, 14, 37], 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 [17].

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 [1] or an in-display fingerprint scanner [11]; a voice assistant may employ arrays of speakers and microphones for audio beamforming [32]; and a VR/AR headset may employ high-resolution displays, requiring powerful GPUs [40]. 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 [13]. 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 [4, 10, 24, 27, 28], is not enough, as it does create multiple virtual instances of the device.

3 Challenges & Design

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

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

Challenge II: path concurrency is feasible but requires environment-awareness. SSE execution is slow due to instruction emulation. To achieve acceptable performance, it is

RIGHTSLINK

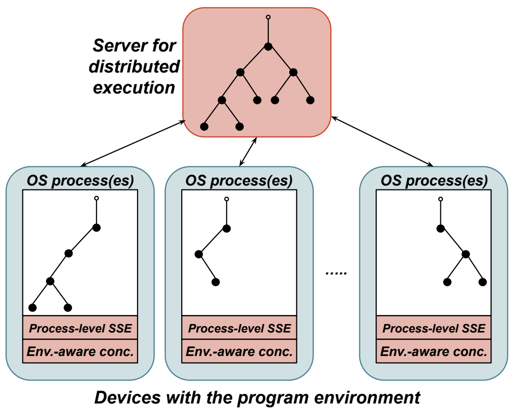


Figure 2. *Mousse design.*

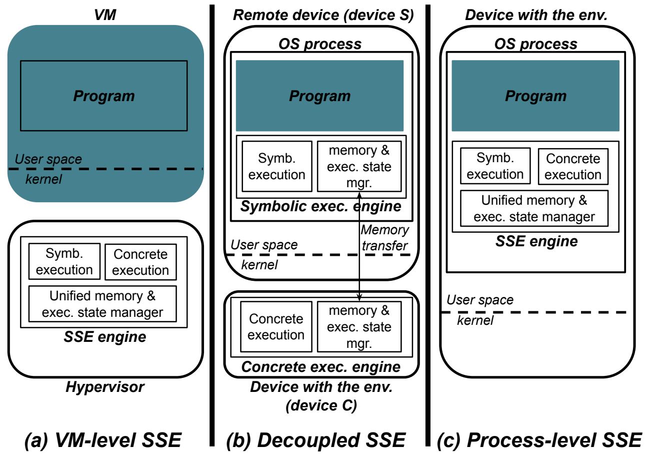


Figure 3. *Different SSE designs.*

important to execute the program paths concurrently. However, the program's interaction with the environment creates a concurrency issue. This is because the environment is stateful (e.g., the state in a device driver or the underlying hardware component). If one program path mutates the environment state, other paths might receive unexpected responses from the environment. Therefore, we introduce *environment-aware concurrency*, a principled 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 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.

4 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^{2}E [17, 18]. Figure 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 (§2.2). 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 [42], Avatar^2 [31], and Symbion [2]. 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 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.

RIGHTSLINK

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 §9.1, 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 [26], RevNIC [15], and SymDrive [35]. 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 [35], 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 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.

4.1 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.

4.2 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 §9.2. 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.

5 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

```
/* 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 4. *A real code example demonstrating the importance of environment-aware concurrency. We have modified and eliminated parts of the code for clarity.*

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

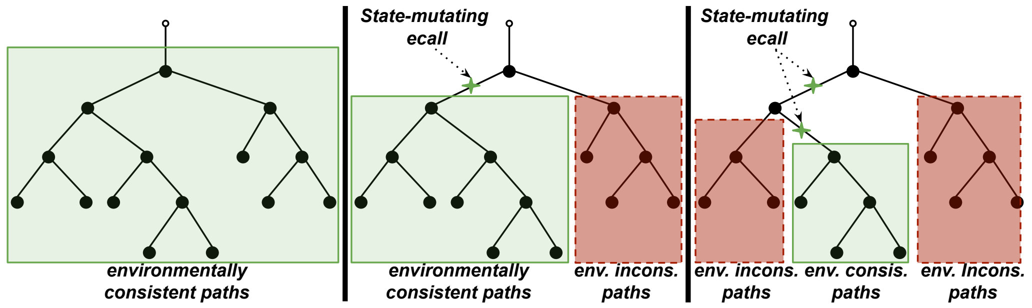


Figure 5. *Environment consistency for concurrent path execution. (Left) All paths to be executed in a device. (Middle) Environmentally consistent and inconsistent paths after a state-mutating ecall. (Right) Paths after the second state-mutating ecall.*

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 §8, 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 5 illustrates this issue. Figure 5 (Left) shows the set of all paths, which are all environmentally consistent in the beginning. Figure 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 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. §6 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 (§6).

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 §9.1, we show that even in the presence of such ecalls, concurrent execution can provide performance benefits. Moreover,

RIGHTSLINK

when such calls are not present, Mousse's solution automatically increases the degree of concurrency.

6 Path Offloading & Distributed Execution

Mousse cannot execute all paths concurrently due to the environment state limitation (§5). 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 (§5), 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.

6.1 Path Offloading

The key component of distributed execution in Mousse is path offloading. Mousse performs path offloading using concolic program inputs. In SSE, one analyzes a program by marking its select inputs (e.g., API inputs or configuration options) as symbolic. During execution, whenever a path needs to be offloaded, Mousse solves the constraints on the path and generates a set of concrete values for program's symbolic inputs. It then offloads these values to the server. When the path is later downloaded by a device for execution, these concrete values can be used to mark the API inputs as concolic variables (§2.1), i.e., concolic inputs. The role of these concolic inputs is to guide the symbolic execution to re-execute the offloaded path from scratch.

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

```
 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 6. *Simple hypothetical program used to demonstrate the offloading strategy in Mousse.*

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 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 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.

6.2 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 §4.2) 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.

7 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.

7.1 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 [30]). 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.

7.2 Target Analyses

We next describe some of the analyses we perform using Mousse. Taking examples from S^2E [17, 18], 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 [5], Amandroid [39], and DroidSafe [25]) 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.

8 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 [17, 18], QEMU (user-mode execution) [8], and KLEE [12]. 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 §7.2). 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) [9] 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`.

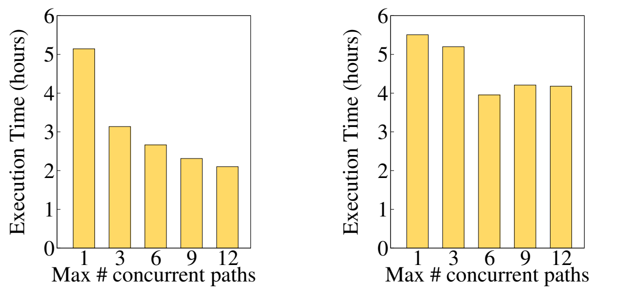
We do not currently support signals, as none of the programs we have analyzed use signals from the environment, e.g., from the driver. Instead, these programs use syscalls (such as `poll` and `select`) to receive notifications. We do, however, support per-process signals, such as `SIGTERM`.

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 (§4.2). 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.

9 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)

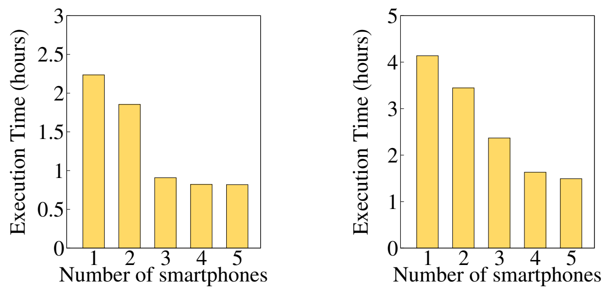
Figure 7. *Impact of environment-aware concurrency on execution time.*

9.1 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 (§7.2) 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 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 (§5). 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 §6, 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.



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

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

Figure 8. *Impact of distributed execution on execution time.*

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 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.*

Distributed execution. Figure 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 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 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 1 shows the results for three services. The table also shows the overall number of paths as well as the offloads due to environment consistency and due to resource constraint. The number of paths varies significantly depending on the API resulting in short (a few minutes) to long (a couple of hours) experiments. Also, the results show that both the environment and resource constraint may result in path offloads.

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 [31]. 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 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.

9.2 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.

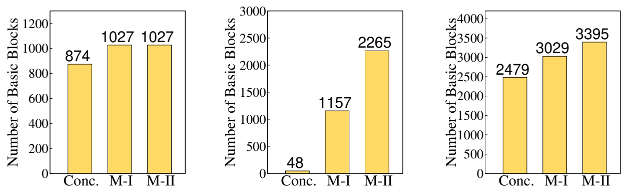
For concrete execution, we try two approaches and report the best one. One is using a known good input to the API that results in deep code coverage. The other is black-box fuzzing, where we try a large number of random inputs to the API and measure the combined coverage.

Figure 9 shows the API coverage for concrete execution and Mousse with its two concretization strategies (§4.2). 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 syscalls with symbolic arguments, both concretization strategies in Mousse achieve the same coverage (Figure 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 §4.2, which one can disable before analysis. However, in the presence of such syscalls, the second concretization strategy achieves higher coverage (Figures 9b and 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.

9.3 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.



(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 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 (§4.2), respectively.*

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.

10 Other Related Work

Charm [36] ports some of the device drivers of mobile devices to run inside VMs. It does so by forwarding the drivers' I/O interactions with the hardware to the mobile device for execution. One may attempt to use Charm along with S^2E to analyze I/O services of mobile devices (indeed, this is the first approach we considered). However, Charm requires some engineering effort to support each device driver (in the order of days). Moreover, it may not port the drivers fully, e.g., it does not support DMA for a GPU driver. Finally, Charm does not virtualize the device, hence S^2E cannot use multiple VMs to interact with it. Since S^2E does not orchestrate interactions with an I/O device hardware (an untamed environment), concurrent execution of VMs would result in unexpected behavior.

Under-constrained symbolic execution, as mainly realized by Under-Constrained KLEE (UC-KLEE) [20, 33, 34], uses symbolic execution to analyze functions with systems code. It does not execute full program paths and simply considers the function arguments to be symbolic. This results in false positives. UC-KLEE therefore provides both automated heuristics and manual methods to add preconditions to the function's input in order to prevent some of the false positives. Mousse, on the other hand, can execute fully-constrained program paths.

DART and SAGE [21–23] automatically generate input for testing of programs by executing them, collecting path constraints, and solving the constraints, an approach otherwise known as concolic testing. Mousse also uses concolic inputs to drive the execution in a desired path (§6.1).

Mayhem [14] and Centaur [29] 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.

AEG [6, 7] uses symbolic execution to automatically generate exploits. Driller [38] uses concolic execution to enhance the performance of fuzzing, an approach referred to as hybrid fuzzing. Both systems model the environment and hence cannot analyze programs with untamed environments.

Qsym [41] is a fast concolic execution engine used for hybrid fuzzing. Qsym avoids taking any path state snapshots and hence re-executes all the paths from scratch using concolic execution. As a result, it can allow the paths to interact with the actual underlying environment. However, Qsym does not provide support for environment-aware concurrency and needs the interactions with the environment to be side-effect free.

Cloud9 [19] distributes symbolic execution over multiple nodes. It, however, does not address the issue of the environment and targets pure symbolic execution (and not selective symbolic execution). Moreover, Cloud9's approach for distributing the execution is different from Mousse's. Cloud9 either uses state copying or state reconstruction to transfer a path from one node to another. State copying is not feasible for programs with untamed environments. State reconstruction is feasible and is indeed what Mousse does. However, Cloud9 uses a bitvector to encode the then/else decisions whereas Mousse uses concolic inputs. Moreover, Cloud9 does not deal with environmentally-forced symbolic variables.

Chipounov et al. define different execution consistency models for SSE, each resulting from different transition points between symbolic and concrete executions and hence resulting in a different set of program paths being analyzed [18]. We note that concretization strategy I in Mousse results in the Strictly Consistent Unit-Level Execution (SC-UE) consistency model whereas concretization strategy II results in the Relaxed Local Consistency (RC-LC) model.

11 Conclusions

We presented Mousse, a system designed to perform selective symbolic execution (SSE) on programs with untamed environments. Mousse provided three novel solutions to deal with such program environments: process-level SSE, environment-aware concurrency, and distributed execution. Through extensive evaluations, we showed that Mousse outperforms alternative solutions in terms of performance and coverage. We also used Mousse to perform various analyses on Android I/O services including bug and vulnerability detection, taint analysis, and performance profiling.

Acknowledgments

The work was supported by NSF Award #1763172. The authors thank Felix Xiaozhu Lin and Zhiyun Qian for their invaluable feedback on an earlier draft of this paper. They also thank the paper shepherd, Marc Shapiro, for his significant help in improving the final version of this paper. In addition, they thank the anonymous reviewers for their insightful comments.

References

[1] 2019. Nokia 9 PureView. https://www.nokia.com/phones/en_int/nokia-9-pureview/. (2019).
[2] 2019. Symbion: fusing concrete and symbolic execution. https://angr.io/blog/angr_symbion/. (2019).
[3] 2020. Mousse source code. https://trusslab.github.io/mousse/. (2020).
[4] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert. 2006. Intel Virtualization Technology for Directed I/O. *Intel Technology Journal* (2006).
[5] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proc. ACM PLDI.*
[6] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. 2011. AEG: Automatic Exploit Generation. In *Proc. Internet Society NDSS.*
[7] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley. 2014. Automatic Exploit Generation. *Commun. ACM* (2014).
[8] F. Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *USENIX ATC, FREENIX Track.*
[9] F. Bellard. 2020. Tiny Code Generator. https://git.qemu.org/?p=qemu.git;a=blob_plain;f=tcg/README;hb=HEAD. (2020).

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

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

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

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

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

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

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

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

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

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

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

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

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

[23] P. Godefroid, M. Y. Levin, and D. Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Commun. ACM* (2012).

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

[25] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. 2015. Information Flow Analysis of Android Applications in DroidSafe. In *Proc. Internet Society NDSS*.

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

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

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

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

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

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

[32] M. Owen. 2018. 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-processor. (2018).

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

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

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

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

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

[38] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proc. Internet Society NDSS*.

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

[40] C. Welch. 2018. 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).

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

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