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

Hsin-Wei Hung
University of California, Irvine
Irvine, CA, USA
hsinweih@uci.edu

Yingtong Liu
University of California, Irvine
Irvine, CA, USA
yingtong@uci.edu

Ardalan Amiri Sani
University of California, Irvine
Irvine, CA, USA
ardalan@uci.edu

ABSTRACT

The Linux kernel is an important part of the Trusted Computing Base (TCB) of a mobile device using the Android OS, making it attractive to attackers. While all vulnerabilities in the kernel are important, those that *are directly reachable by untrusted programs* pose a grave threat. This paper introduces Sifter, a solution for protecting security-critical kernel modules, i.e., those modules that are directly exposed to untrusted programs. Sifter's key approach is the use of fine-grained, highly-selective filters to reduce the attack surface of these kernel modules and make their vulnerabilities unreachable for untrusted programs. The key observation in Sifter is that there are rich patterns in how legitimate programs issue syscalls to these kernel modules; thus, one can generate filters that only allow such syscall patterns, and as a result mitigate vulnerabilities (including zero-day ones) that could only be exploited by the use of unorthodox syscall patterns.

We report a prototype of Sifter and use it to generate filters for two security-critical kernel modules used in many mobile devices: Qualcomm KGSL GPU device driver and Binder IPC. Our detailed study and evaluation of 41 recent CVEs in these two modules show that Sifter is capable of mitigating about half of all syscall-triggered vulnerabilities without *a priori* knowledge of these vulnerabilities. Moreover, our evaluation shows that when using an adequately large number of legitimate programs to generate the filter policies for a given module, the filter's false positive rate goes to 0%. Finally, our experiments with these filters show that, despite numerous fine-grained checks on syscalls, Sifter adds a very small or negligible performance overhead to real programs and incurs a very small amount of energy consumption.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**; **Mobile platform security**; **Systems security**.

KEYWORDS

Syscall filtering, attack surface reduction, Linux kernel security, Android security, eBPF

ACM MobiCom '22, October 17–21, 2022, Sydney, NSW, Australia

ACM ISBN 978-1-4503-9181-8/22/10.
https://doi.org/10.1145/3495243.3560548

ACM Reference Format:
Hsin-Wei Hung, Yingtong Liu, and Ardalan Amiri Sani. 2022. Sifter: Protecting Security-Critical Kernel Modules in Android through Attack Surface Reduction. In *The 28th Annual International Conference on Mobile Computing and Networking (ACM MobiCom '22), October 17–21, 2022, Sydney, NSW, Australia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3495243.3560548

1 INTRODUCTION

The Linux kernel is an important part of the TCB of a mobile device using the Android OS. To compromise these devices (e.g., through privilege escalation), adversaries seek to exploit vulnerabilities in the kernel. Kernel exploits are very powerful; for example, a successful privilege escalation attack may allow malware to perform arbitrary operations. Kernel modules, e.g., device drivers, are especially concerning as they are often developed by third-party developers and contain many bugs (e.g., according to a report from Google on Android [1]).

However, not all vulnerabilities are created equal. More specifically, *those vulnerabilities that can be reached from untrusted programs pose a grave threat as they can be exploited by any program, e.g., any Android app.* Fortunately, Android makes most kernel modules inaccessible to untrusted programs, which greatly helps address this concern. For example, programs cannot directly interact with the camera and audio device drivers. They instead have to use these drivers through proxies, i.e., Android OS services with the unique permission to use these modules [2, 3].

This approach, while effective, cannot be applied to all kernel modules. This is because it impacts performance by adding another layer of indirection. Therefore, in Android, some performance-sensitive modules are still exposed to and directly used by untrusted programs. We refer to these modules as *security-critical* since their vulnerabilities are reachable by untrusted programs. Important examples of security-critical modules are Binder IPC, GPU device driver, and WiFi device driver. Indeed, an analysis from Google of bugs observed in Android's kernel shows that the majority of bugs reachable by untrusted program reside in these modules (excluding the `debugfs` and `perf` kernel modules as they can be simply disabled on production devices) [1].

Our goal in this paper is to protect security-critical kernel modules in Android. Our idea to is to mitigate vulnerabilities in these modules by narrowing the attack surface through *system call (syscall) filtering*. Using syscalls is one of the key methods malware uses to exploit a vulnerability in these modules (§7). A syscall filter is deployed in the kernel. It analyzes every syscall issued by an untrusted process to these modules and decides whether to allow or reject it according to some policies. The hope is that, even though the vulnerability is still present, malware will not be able to exploit

the vulnerability since it simply cannot trigger it using syscalls because of the filter.

We present Sifter, the first system designed to generate fine-grained, highly-selective filters for *mitigating vulnerabilities in security-critical kernel modules in Android*. Sifter is motivated by two simple observations. First, syscalls needed to interact with a kernel module follow a rich set of patterns, encoded in the module itself, often through a set of `ioctl` syscalls. Second, to exploit a vulnerability in a module, malware often issues syscalls with meticulously-crafted arguments and sequences, which are not normally used by legitimate programs. Therefore, by rejecting bizarre-looking syscall patterns, Sifter can make it impossible for malware to trigger a large number of kernel vulnerabilities in security-critical modules even if the module is directly exposed to malware.

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

To achieve these goals, Sifter leverages two recent advances related to Linux. First, it leverages eBPF programming, which is widely available in modern Android-based mobile devices, to securely deploy *stateful* syscall filters and to *safely access the program memory*. Second, Sifter leverages *syscall definition templates* available in syzkaller [4], the state-of-the-art kernel fuzzer, to provide the required domain knowledge for kernel modules.

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

We answer two important research questions:
Q1. Can effective filtering policies be automatically generated? Sifter automatically generates and enforces three types of filtering policies. First, it restricts the syscall arguments depending on the type of argument, e.g., a length or a flag argument. Second, it serializes non-sleeping module-related *operations* (i.e., one or multiple syscalls used together to perform an operation). Third, it filters out legacy and deprecated operations never used by legitimate programs. We address several challenges in constructing these filters including dealing with argument outliers and identifying non-sleeping indivisible syscall sequences forming the aforementioned operations.

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

In addition, using a large number of programs (and using disjoint sets of programs for training and testing), we demonstrate that Sifter filters do not break the execution of legitimate programs, resulting in 0% false positive rates, as long as an adequate number of programs are used for training.
Q2. Do such filters introduce prohibitive performance overhead? As mentioned earlier, these security-critical kernel modules are performance-sensitive (indeed, as discussed, that is why they are directly exposed to untrusted programs). We report a performance evaluation of Sifter. Our evaluation shows that Sifter adds a modest performance overhead to micro-benchmarks. More importantly, it *incurs a very small or negligible performance overhead to real programs*. We also show that Sifter incurs a very small amount of energy consumption.

These results indicate that Sifter is a practical method to effectively curb the attack surface of security-critical kernel modules and mitigate many vulnerabilities.

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

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

Figure 1: *Simplified code illustrating CVE-2017-9682.*

2 OVERVIEW

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

2.1 Motivating Examples

CVE-2017-13162. To exploit this vulnerability, a malicious program must first issue an `mmap` syscall to Binder with a size that is smaller than `BINDER_MIN_ALLOC`. However, this argument value is never used by legitimate programs. This is because legitimate programs interact with Binder using the library `libbinder`, which passes a fixed size for `mmap`, one that is larger than `BINDER_MIN_ALLOC`.
CVE-2017-9682. Figure 1 shows this CVE, which is a Use-After-Free (UAF) vulnerability in the KGSL GPU driver. When a program creates a draw context by calling the `ioctl` syscall with the

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

2.2 Goals

We pursue three goals in Sifter.

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

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

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

2.3 Feasibility

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

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

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

Fortunately, eBPF (extended BPF) support has been added to the Linux kernel (and hence Android) recently [5] and it addresses the aforementioned limitations. eBPF has a richer (compared to BPF) set of instructions and supports a persistent data structure across syscalls (called *map*). In addition to the greatly extended capability, it also improves performance due to the use of just-in-time compilation. For enhanced security, eBPF programs are verified before being loaded into the kernel.

Indeed, there are several frameworks that can deploy eBPF programs in the kernel today. Since its introduction, *kernel tracing mechanisms* such as kprobe, tracepoint, perf event, and raw tracepoint have gradually supported eBPF [6]. In fact, our own tracer in Sifter leverages eBPF with tracepoint, as we will discuss. In 2018, integration of eBPF with seccomp was supported [7], although it has not been yet integrated into the mainline Linux. Finally, in 2020, integration of eBPF with Linux security hooks, also known as Kernel Runtime Security Instrumentation (KRSI), has been supported and added to the mainline Linux (starting version 5.7) [8–10].

Use of eBPF can enable us to achieve the first two goals in §2.2. Since all their policies are implemented in eBPF programs, Sifter filters can be deployed in practice using either seccomp/eBPF or KRSI. We use seccomp/eBPF in our prototype, but also provide a discussion of how Sifter can be deployed using KRSI (§9).

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

We observe that kernel fuzzers also require the same domain knowledge to be able to effectively fuzz a kernel module. They rely on *syscall definition templates* to achieve this goal. As a result, security analysts have developed a large number of such templates. More specifically, syzkaller [4], the state-of-the-art kernel fuzzer provides such templates for 170 kernel modules (as of December 2021) including those specific to Android. Therefore, we reuse these templates in Sifter to eliminate the need for domain knowledge provided manually by the analyst. This enables us to achieve the last goal in §2.2.

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

We note that syzkaller might not have a template for a kernel module. In this case, the analyst needs to develop the template. This involves studying the syscall interface of a kernel module

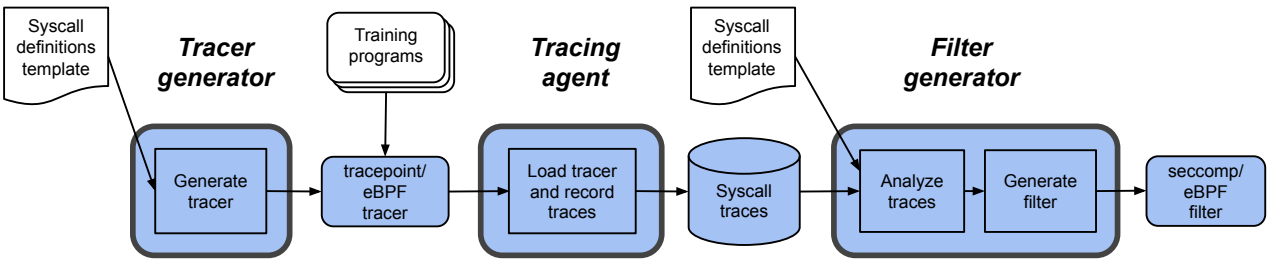


Figure 2: ***Sifter's workflow. The blue shapes are part of Sifter and the white ones are external.***

and then defining the syscalls along with their arguments using syzkaller's syscall description language. However, given the popularity of syzkaller and other kernel fuzzers [11–19], we believe that in the near future, many kernel modules will have their templates available. As an example, at the time of this work, syzkaller's main repository did not have a template for the KGSL GPU driver used in our prototype. However, our own research group had previously developed the template for this module as part of a fuzzing research project [20]. Therefore, we borrowed the template from that project and added minor improvements. This significantly reduced our manual effort.

2.4 Remaining Research Questions

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

Q1. Can effective filtering policies be automatically generated? We show that we can automatically extract three different types of policies (i.e., argument limiting, operation serialization, and operation deprecation policies) related to a security-critical kernel module by observing how legitimate programs issue syscalls to the module (§4 and §5). We also show that our automatically generated policies can mitigate about half of all syscall-triggered vulnerabilities (§7). Moreover, we show that they do not introduce false positives (§8.1).

Q2. Do such filters add prohibitive performance overhead to legitimate programs? We demonstrate that our filters introduce very small or negligible performance overhead to real programs (§8.2). Moreover, they incur a very small amount of energy consumption (§8.3).

3 THREAT MODEL

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

4 WORKFLOW

Figure 2 shows the workflow of Sifter. There are three components in Sifter: the *automatic syscall tracer generator*, the *tracing agent*, and the *automatic filter generator*.

Imagine a security analyst who wants to use Sifter to generate a filter for a security-critical kernel module in Android. In the first step, the analyst uses the tracer generator to generate a tracer for that kernel module. To do so, the analyst feeds the syscall definition template of that module (§2.3) to the tracer generator, which automatically generates the source code of a tracepoint/eBPF syscall tracer.

In the second step, the analyst uses the tracing agent to collect traces of syscalls from a set of *training programs*. The analyst needs to choose a representative and comprehensive set of training programs in order to capture all legitimate and expected syscall patterns (see Table 2 in §6.5 for the detailed list of the training programs in our prototype). To perform this step, the analyst uses the agent to deploy the tracer, execute the training programs, and collect the syscall traces. The tracing agent automatically hooks the tracepoint/eBPF tracer program to syscall entry and return points. The tracer logs the syscall number, the arguments in the registers and the timestamp for all syscalls into an eBPF map. When the syscall is for the kernel module of interest, the tracer will additionally perform a deep copy of arguments from the user space memory to additional eBPF maps.

In the third step, the analyst uses the filter generator to generate the filter. The generator reads the syscall traces, performs a series of analyses to extract filter policies (§5), and then generates a seccomp/eBPF filter to enforce the policies. The filter generator also uses the syscall definition template to generate code to parse syscall arguments at runtime.

In the fourth step, the analyst evaluates the false positive rate of the filter (§8.1). If not zero, the analyst chooses additional programs for training and goes back to the second step. This continues until the false positive rate reaches zero.

In the final step, the analyst deploys this filter into a target mobile device. As mentioned in §2.3, our prototype currently leverages seccomp to deploy the eBPF filter. In this case, the filter is deployed selectively for operating system processes. More specifically, similar to existing seccomp filters, the filter needs to be installed for the process before the execution of any untrusted code in that process. Given that a seccomp filter cannot be disabled once installed, the untrusted code cannot escape the filter. As also mentioned in §2.3, we can also use KRSI to deploy our filters. In that case, the filter will be deployed for a specific module (see §9 for a discussion on this matter). Figure 3 illustrates the filter in action.

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

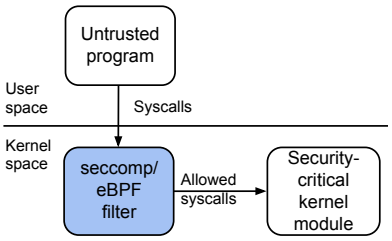


Figure 3: ***The Sifter's filter limits syscalls of untrusted programs.***

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

5 FILTER POLICIES

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

5.1 Argument Limiting Policy

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

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

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

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

We have found two outliers in all of the programs that we tested (§6.5) and we have manually confirmed that each of the detected outliers are indeed incorrect usages of a syscall.

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

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

5.2 Operation Serialization Policy

A large number of vulnerabilities in the kernel are race conditions (e.g., 43% of all syscall-triggered vulnerabilities that we study in §7). To exploit a race condition, a malicious program often needs to issue syscalls simultaneously from multiple threads to cause a race. A straightforward approach to mitigating such vulnerabilities is serialization, which eliminates concurrency.

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

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

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

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

syscalls. Hence, our operation serialization policy is a stricter policy compared to the strawman syscall serialization policy.

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

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

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

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

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

After applying this algorithm to syscall traces from real programs (§6.5), we do find some interesting sequences, representing the aforementioned high-level operations used by programs. For example, we find a short `ioctl(fd, IOCTL_KGSL_GPUOBJ_ALLOC)`→ `mmap(addr, PROT_READ|PROT_WRITE, MAP_SHARED, ...)` sequence, which suggest that the GPU object allocation operation always maps the GPU memory into user space after creating a memory entry in the GPU driver. We also find some longer operations. For example, we find an operation with 8 syscalls, all `IOCTL_KGSL_GETPROPERTY` but with different values for a *flag* argument, `type`: 0x13→0x17→0x18→0x15→0x19→0x20→0x1a→0x1b. This operation is used during the GPU initialization phase.

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

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

Table 1: ***Information of filters generated by Sifter. P_{Arg} is the number of argument limiting policies, $P_{Op.S}$ is the number of operations in the operation serialization policy, and $P_{Op.D}$ is the number of syscalls allowed by the operation deprecation policy. #Inst. is the number of eBPF instructions.***

5.3 Operation Deprecation Policy

Kernel modules often have several legacy and old syscalls not used by legitimate programs. However, these syscalls can be used by malware for exploits [1, 21]. We develop a policy that disallows such syscalls.

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

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

6 IMPLEMENTATION & PROTOTYPE

We implemented Sifter in Go and C++ in 6000 LoC. We developed the seccomp/eBPF filtering mechanism based on [7], and integrated it with the Linux kernel 4.9 for Android on Google Pixel 3. As discussed in §9, it is feasible to deploy Sifter's filters using the KRSI framework as well. We note that eBPF is widely available on modern Android-based mobile devices, i.e., those with Linux kernel version 4.9 or higher.

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

Table 1 provides some information about the two filters we have constructed. The table shows the number of rules of different policies of the generated filters. It also reports the number of eBPF instructions along with the total size of the maps (where we calculate the size of a map by multiplying the size of an entry by the number of entries).

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

6.1 Checking File Descriptors

Sifter enforces its policies per opened file descriptor. When a program needs to use a module, it first opens the module, which returns a file descriptor. All following syscalls are then issued by passing

the file descriptor. We enforce the policies per file descriptor because kernel modules maintain different state information for each opened file descriptor.

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

6.2 Deploying Complex Filters using eBPF

A filter generated by Sifter is complex since it walks through deep memory arguments and maintains the state of the syscall sequences. In our first prototype, we implemented the filter in a single eBPF program. However, that exceeded the complexity limit of the eBPF verifier and thus was rejected. This was because the verifier could not traverse every possible path in the program to check its safety. To address this challenge, we break the filter into several smaller filters: multiple argument limiting filters, one filter to reject single-syscall deprecated operations (and to reject syscalls not approved by at least one argument-limiting filter), and one filter to serialize the operations and to reject deprecated multi-syscall operations. These filters use an eBPF map to share information.

6.3 Preventing Collusion Attacks

When analyzing syscall traces to extract operations (§5.2), we look at syscalls triggered by each thread separately. This is because in a well-behaved program, there is no reason to use more than one thread to issue a set of dependent syscalls to realize an operation together. However, when enforcing the operations (i.e., sequence of syscalls), we cannot simply enforce them per thread. If we do, malware can use colluding threads or processes to bypass the filter. We devise solutions to prevent such collusions, as discussed next.

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

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

6.4 Preventing TOCTOU Attacks

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

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

6.5 Training Programs

We used a diverse set of training programs for our filters. For Binder IPC and the GPU driver, we record syscalls issued by 60 Android apps downloaded from the Google Play store. The training programs are selected from the top charts of various categories based on popularity and review score. They include popular video streaming apps, shopping apps, games, social media apps, and tools. Table 2 shows the detailed list of the training programs. For each of them, we start the tracing agent and manually interact with the app for 30 minutes with the goal of exercising as much functionality as possible. The syscall tracing results are later used to generate filters. This manual interaction with the app is currently the most time-consuming aspect of our system. However, we think the manual effort is acceptable since there are only a limited number of security-critical kernel modules that programs can directly access in Android. Besides, the two security-critical kernel modules we analyzed are widely available: Binder is available on all Android devices and Qualcomm GPUs have a significant market share. Therefore, the effort could benefit many devices in the wild with the potential of mitigating zero days. One option to automate the manual trace collection is to use a monkey, e.g., the Android UI/Application exerciser [22]. However, a monkey might not be able to invoke as many functionalities of the app as we manually do since it often gets stuck at certain UI elements. We think that tuning a monkey to generate syscall traces with high coverage needed by Sifter will be a useful future work.

7 EFFECTIVENESS STUDY

We present a study of 41 Linux CVEs from the Binder IPC module and the Qualcomm KGSL GPU device driver used in many mobile devices. These CVEs are all that we found for the period of 2016-2020. We collect these CVEs from the MITRE CVE database and the Android security bulletin. We have two key goals in this study. First, we use the study to show empirically that a large number of vulnerabilities are triggered by syscalls and hence can, at least theoretically, be mitigated by syscall filters. Second, we use the study as a framework to evaluate the effectiveness of Sifter.

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

Table 2: ***Training programs used for generating syscall policies.***

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

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

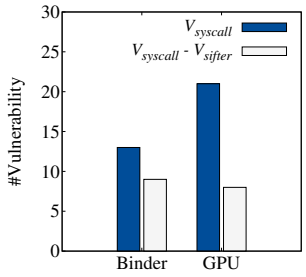


Figure 4: ***The numbers of triggerable vulnerabilities in $V_{syscall}$ before and after applying filters.***

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

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

The number of triggerable vulnerabilities in $V_{syscall}$ before and after applying filters is illustrated in Figure 4 and the detail breakdown is shown in Table 3. Our analysis shows that 18 of the vulnerabilities are mitigated using Sifter (V_{sifter}). They account for about 51% of syscall-triggered vulnerabilities ($V_{syscall}$).

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

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

First, the argument limiting policy mostly mitigates Out-Of-Bounds (OOB) vulnerabilities. Figure 5 shows one such vulnerability, CVE-2016-2468, in the Qualcomm GPU driver. During GPU memory allocation, the driver updates `sglen` in the loop at line 12 based on `len`, which gets directly assigned from a user-controllable argument of the function, `size`, earlier at line 8. Since the datatype of `len` is a signed integer, assigning it with a value greater than 0x80000000 will result in a negative `len`. As as result, `sglen` never gets updated in the loop, and when being used to index an array at line 16, OOB access happens. However, during the training, it never exceeds 0x80000000. The argument policy generated by Sifter limits its upper bound to around 0x1E000000, therefore preventing it from being exploited.

Second, the operation serialization policy in our analysis mostly mitigates race condition vulnerabilities, especially UAFs. We explain how it works using CVE-2018-13905 as an example, which is illustrated in Figure 6. In the Qualcomm GPU driver, the book-keeping of sync-sources is done by using `idr`. The user can destroy a sync-source by calling the `ioctl` syscall with an argument specifying the ID. The driver first invokes `kgsl_ioctl_syncsource_destroy` at line 14. It then looks up the sync-source from the `idr`, calls `kgsl_syncsource_cleanup` to remove it from the `idr`, and then releases the sync-source. While every operation on `idr`, namely

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

Figure 5: *Simplified code illustrating CVE-2016-2468.*

```
1  static void kgsl_syncsource_cleanup(
2      struct kgsl_process_private *private,
3      struct kgsl_syncsource *syncsource) {
4    ...
5    spin_lock(&private->syncsource_lock);
6    if (syncsource->id != 0) {
7      idr_remove(&private->syncsource_idr, syncsource->id);
8      syncsource->id = 0;
9    }
10   spin_unlock(&private->syncsource_lock);
11   ...
12 }
13
14 kgsl_ioctl_syncsource_destroy(
15     struct kgsl_device_private *dev_priv,
16     unsigned int cmd, void *data) {
17   ...
18   spin_lock(&private->syncsource_lock);
19   syncsource = idr_find(&private->syncsource_idr,
20                         param->id);
21   spin_unlock(&private->syncsource_lock);
22   ...
23   if (syncsource == NULL)
24     return -EINVAL;
25   kgsl_syncsource_cleanup(private, syncsource);
26   return 0;
27 }
```

Figure 6: *Simplified code illustrating CVE-2018-13905.*

`idr_find` and `idr_remove`, seems to be protected by a `spinlock`, the destroy process is not atomic. When one thread is trying to destroy a sync-source, after it finds the sync-source and before removing it, another thread might acquire a reference to the same sync-source. After it is destroyed by the first thread, access to the dangling pointer in the second thread will cause a UAF.

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

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

Non-mitigated vulnerabilities. An example of these non-mitigated vulnerabilities is CVE-2020-0041 in the Binder IPC. The vulnerability is caused by an incorrectly calculated variable used for checking the boundary. This allows an attacker to violate the rules of how binder objects should be placed in the binder transaction buffer. The binder transaction buffer may hold a series of objects with hierarchical relationships and requires specific logic to validate its correctness. Sifter unfortunately cannot mitigate this vulnerability.

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

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

7.1 Mimicry Attacks

One might wonder about Sifter's defense against mimicry attacks [23]. That is, one might wonder whether Sifter's filters can be bypassed by (1) crafting the syscall arguments, (2) substituting some syscalls in the syscall sequence with equivalent syscalls, or (3) inserting no-op syscalls or delays between the syscalls in an operation to deceive the filter. (These are all types of mimicry attacks that we could think of.) In our analysis, Sifter is secure against such mimicry attacks, as discussed next.

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

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

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

8 EVALUATION

8.1 False Positive Analysis

Sifter's dynamic approach in generating syscall policies could yield stricter filters when compared to statically analyzing syscall usages of the kernel module from applications or libraries. However, on the other hand, it has a higher chance of accidentally blocking or

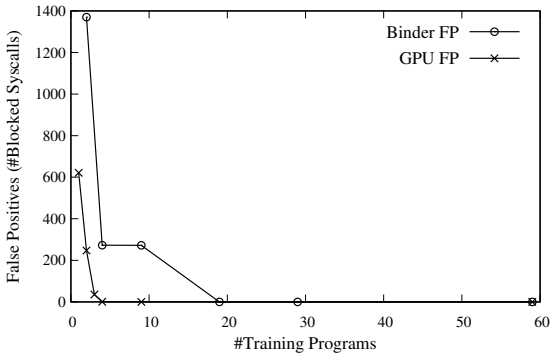
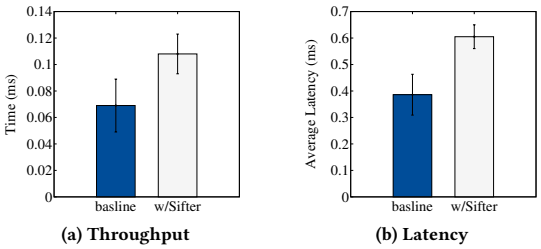


Figure 7: ***False positives of filters trained and tested using different numbers of training programs.***



(a) Throughput **(b) Latency**

Figure 8: ***Performance overhead of Sifter on Binder micro-benchmarks.***

reporting (depending on the violation policy discussed in §4) a legitimate syscall, i.e., a false positive. In this section, we demonstrate that, with enough data, our filters can achieve a 0% false positive rate.

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

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

The result, illustrated in Figure 7, shows that initially, filters of both modules have significant amount of false positives, which are contributed by all policies due to the lack of training data. However the false positive rate is reduced dramatically with increasing the training data set size. For GPU driver, we can reach 0% false positives after using 9 programs for training. For Binder, it needs more training data to reach 0% false positive. The false positives drop to 0% after we increase the number of training programs to 19. Note that the required amount of programs for different kernel modules may vary, and the train-and-test method can serve as a indication of whether the training data is enough.

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

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

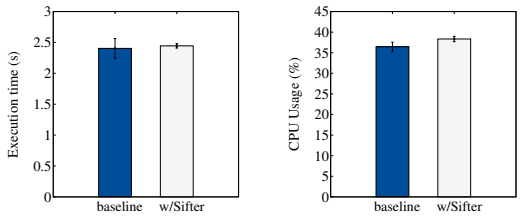
It is worth mentioning that although we do not change the aforementioned thresholds in our evaluation for both modules, they theoretically can differ for kernel modules even for the same type of arguments as they carry different semantics. For the threshold used in operation serialization policy, it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

8.2 Performance

We measure the performance overhead of Sifter on Binder IPC and the Qualcomm KGSL GPU driver. To evaluate the performance overhead introduced by Sifter, we install the filters generated by Sifter on selected benchmark programs including both synthetic benchmarks (i.e., micro-benchmarks) and real programs (i.e., macro-benchmarks). Then, we report the execution time as well as the CPU usage. We run each experiment ten times and show the mean and standard deviation (the latter using error bars).

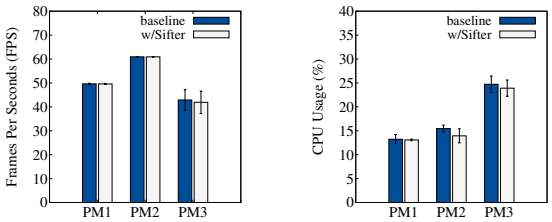
Binder. To evaluate the impact of Sifter on Binder, we choose two micro-benchmarks, `BinderThroughput` and `schd-dbg`, and one macro-benchmark, `SurfaceFlingerStress`. `BinderTroughput` measures the average time consumed by Binder transactions of various sizes, and `schd-dbg` measures the average latency of Binder transactions. The results of the micro-benchmarks, shown in Figure 8, show that Sifter increases the transaction time by 49%. The latency micro-benchmark shows transactions on average spend 56% more time in the kernel.

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



(a) Performance (b) CPU usage

Figure 9: ***Performance overhead of Sifter on Binder macro-benchmark (SurfaceFlingerStress).***



(a) Performance (b) CPU usage

Figure 10: ***Performance overhead of Sifter on GPU macro-benchmarks (PM1: Passmark 3D simple, PM2: Passmark 3D complex, PM3: Passmark OpenGL ES).***

of its syscalls are Binder `ioctl`s. Figure 9a shows the execution time of `SurfaceFlingerStress`, which is a test program stressing `SurfaceFlinger` by creating and destroying surfaces for 1,000 times in 10 threads, respectively. Therefore, we use it to simulate a Binder IPC intensive application. Although Sifter introduces a noticeable overhead to Binder transactions, it only slows down the extremely IPC-intensive program by 1.7% and increases the CPU usage by 5% (Figure 9b).

GPU driver. To evaluate the performance overhead of the filter on GPU in real world scenarios, we apply the filter to three 3D benchmarks. We measure the average CPU usages and the frames-per-second (FPS) during the testing session. Figure 10a shows the FPS of the three macro-benchmarks. The performance overhead introduced by the syscall filter in terms of FPS is almost negligible whether the benchmark programs stress the GPU or not. The differences of FPS are within the measurements' error margins. Figure 10b shows the CPU usages of the benchmarks before and after installing the filter. The CPU usage increase is also almost unnoticeable in the first GPU benchmark. In the second and third benchmarks, the CPU usage of the program with filter is slightly lower than the one without by 4.4% and 3.3%, respectively. (We suspect that the small amount of idle time caused by the locks in our serialization policy results in the slightly lower CPU usage when the filter is used.) This again shows that the performance overhead introduced by Sifter is almost negligible in real world scenarios.

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

8.3 Energy Consumption

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

8.4 Training Time

The training time of a filter is proportional to the training data set size. When using the largest data set, which comprises 60 30-minute traces with a total size of 131 GB, it takes Sifter two hours to analyze the traces and generate a syscall filter on an Intel Xeon CPU E5-2697v4 machine using a single core.

9 DISCUSSIONS

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

Deploying Sifter filters beyond Android. Sifter is built on top of eBPF, which is originally developed for Linux. Thus, Sifter is not limited to Android and can be used for all devices using the Linux kernel. Moreover, due to the popularity of eBPF and its ability of extending kernel functionality, Microsoft is also adopting it in Windows [24]. Therefore, Sifter may be applicable to a wider range of mobile devices in the future, such as laptops and embedded devices.

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

10 RELATED WORK

Syscall filtering. There is a long line of work on syscall filtering frameworks [25–32]. These filtering frameworks use different hooks, which give them different capabilities and performance implications. ptrace-based approaches are generally slower due to their inability to trace syscalls selectively [28]. Mbox [27] utilizes seccomp/BPF to only monitor syscalls of interest and then invoke the ptrace-based tracer to query the policy engine, which greatly improves performance. However, they all require a policy engine running in the user space. As a result, to check a syscall, at least two context switches need to be made. In addition, more ptrace

syscalls are needed to inspect the user space memory or register content in the tracee. These context switches make the approach less efficient. Compared to Mbox, Sifter's eBPF approach does not result in context switches since the eBPF filter runs within the same context as the syscall-calling process. User memory probing helper functions also eliminate the need for additional context switches.

SELinux [33] also has the ability to filter syscalls. With extended permission access vector rules, it can also check some arguments of syscalls. For example, the command argument of `ioctl`. However, it cannot probe and check deeper into in-memory arguments. Without the ability to inspect in-memory arguments, 4 out of 5 vulnerabilities mitigated by Sifter's argument limiting policy would not have been mitigated. Also, it does not provide memory to construct stateful policies.

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

Kernel specialization. To reduce the attack surface of the kernel, kernel specialization methods restrict access to the kernel code for target applications by debloating it. KASR [34] first uses offline training stage to generate the kernel code usage database for a specific application in a trusted hypervisor. It then uses runtime enforcement to remove access permission to unused code of executable and selectively activate the corresponding used code. FaceChange [35] differs from KASR by supporting multiple applications running together. Using a hypervisor, it creates specialized kernel views for each of the target applications and switches between them. MULTIK [36] also supports multiple applications by orchestrating multiple kernels specialized for applications, but avoids the overhead of virtualization and runs natively on the system. Compared to these three schemes, SHARD [37] significantly reduces the attack surface by specializing at both the application and system call levels and strictly enforcing debloating in a more fine-grained way. A limitation of kernel specialization is that it cannot mitigate a vulnerability if the vulnerable code is needed and used by the application. Sifter, however, can mitigate such vulnerabilities.

Policy generation. To generate the syscall filtering policy without domain knowledge, Sifter takes the approach of inferring it from a set of programs, which is similar to systrace [31]. However, to better capture the behavior of programs, systrace translates syscalls with arguments to higher-level information with semantics. This approach requires the developer to manually define the semantics and craft the translation logic. Although Sifter lacks the complete semantics, it does not blindly restrict syscalls arguments to known values. With the syscall definition provided by syzkaller, we define different argument policies for different type of arguments. Moreover, Sifter tries to automatically identify higher-level operations consisting of multiple syscalls.

SPOKE [38] is a policy generation framework for SEAndroid. It extracts the domain knowledge from functional tests since they could carry rich semantics. Access patterns from multiple layers (e.g., Dalvik layer, native layer, and kernel layer) are collected, correlated, and noise-filtered to form a knowledge base. In contrast, Sifter collects syscall usage traces from real programs. Abhaya [39] uses interprocedural static analysis to generate syscall policies automatically. They do not however generate filters for kernel modules.

Offline intrusion detection. A long line of research has been conducted on detecting intrusion or anomaly [40–49]. Syscall-based intrusion detection [40–42, 44, 47, 48] works by first capturing normal program syscall behaviors using data models [40–42], classification algorithms [44] or machine learning techniques [49]. Then, at the next stage, it utilizes a syscall interposition mechanism to track syscalls of an interesting process and check against the model. Although some of the techniques are shown to be effective, they often run the analysis offline (i.e., syscall traces are analyzed after being made) in the user space due to their complexity. Therefore, it leaves a vulnerability window for an adversary to perform an attack before being caught. Sifter does not have this problem because it directly deploys its policies in the kernel and check syscalls against those policies at runtime. In Sifter, we show that policies we use incur little performance overhead and they stop a large number of CVEs from being exploited without *a priori* knowledge of these vulnerabilities.

Android Malware detection. In Android, app marketplaces such as Google Play, Amazon Appstore, and T-Market perform malware detection to prevent spreading them. Similar to offline intrusion detection, a model is trained using different techniques and scanning is performed when a developer submits an app. APIChecker [50] used by T-Market takes a deny-list approach by training a API-usage-based classification model using known good apps and malicious apps. In Sifter, we take an allow-list approach: we only use a set of good apps and assume zero-days are not yet known. In addition, Sifter is a runtime defense compared to the one-time scan. A malicious app can evade "scan once and allow" approaches if the analysis fails to cover the code containing malicious API calls, or the malicious app is able to detect the sandbox environment and stay dormant during the analysis. In fact, [51] has demonstrated that creating an undetectable sandbox environment is extremely hard. Therefore, our runtime approach that does not require knowledge of vulnerabilities is orthogonal to APIChecker.

11 CONCLUSIONS

We presented Sifter, a syscall filtering solution to mitigate vulnerabilities in security-critical kernel modules in Android. Security-critical modules are those that are directly exposed to untrusted programs. Sifter's filters enforce the syscall patterns used by legitimate programs, observed in a training phase and distilled into three different types of policies. Through a comprehensive vulnerability study and evaluation, we showed that Sifter can mitigate about half of all syscall-triggered vulnerabilities. In addition, we showed that Sifter's filters have 0% false positive rate. Finally, our performance evaluation showed that Sifter adds a very small or negligible performance overhead to real programs and incurs a very small amount of energy consumption.

ACKNOWLEDGMENTS

The work was supported in part by NSF Awards #1763172 and #1846230 as well as Google's 2020 Android Security and PrIvacy REsearch (ASPIRE) Award. The authors thank the anonymous reviewers and shepherd for their insightful comments.

REFERENCES

[1] Jeffrey Vander Stoep. Android: Protecting the Kernel. In *Linux Security Summit (LSS)*, 2016.

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

[3] Yingtong Liu, Hsin-Wei Hung, and Ardalan Amiri Sani. Mousse: A System for Selective Symbolic Execution of Programs with Untamed Environments. In *Proc. ACM EuroSys*, 2020.

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

[5] Alexei Starovoitov. BPF syscall, maps, verifier, samples, llvm. https://lwn.net/Articles/609433/, 2014.

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

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

[8] Jonathan Corbet. The 5.7 kernel is out. https://lwn.net/Articles/821829/, 2020.

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

[10] Jake Edge. Kernel runtime security instrumentation. https://lwn.net/Articles/798157/, 2019.

[11] Dave Jones. Triforce linux syscall fuzzer. https://github.com/nccgroup/TriforceLinuxSyscallFuzzer, 2016.

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

[13] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. DIFUZE: Interface Aware Fuzzing for Kernel Drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2017.

[14] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 167–182, 2017.

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

[16] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. HFL: Hybrid Fuzzing on the Linux Kernel. In *Proc. Network and Distributed Systems Security Symposium (NDSS)*, 2020.

[17] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (S&P). IEEE*, 2019.

[18] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. KRACE: Data race fuzzing for kernel file systems. In *2020 IEEE Symposium on Security and Privacy (S&P). IEEE*, 2020.

[19] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V Krishnamurthy, and Nael Abu-Ghazaleh. SyzVegas: Beating Kernel Fuzzing Odds with Reinforcement Learning. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2741–2758, 2021.

[20] Seyed Mohammadjavad Seyed Talebi, Zhihao Yao, Ardalan Amiri Sani, Zhiyun Qian, and Daniel Austin. Undo Workarounds for Kernel Bugs. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2381–2398, 2021.

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

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

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

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

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

[26] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. In *Proc. Network and Distributed Systems Security Symposium (NDSS)*, 2004.

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

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

[29] Giovanni Russello, Arturo Blas Jimenez, Habib Naderi, and Wannes van der Mark. Firedroid: Hardening security in almost-stock android. In *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC)*, pages 319–328, 2013.

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

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

[32] DeMarinis, Nicholas, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. Sysfilter: Automated system call filtering for commodity software. In *In 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[33] Stephen Smalley, Chris Vance, and Wayne Salamon. Implementing selinux as a linux security module. *NAI Labs Report*, 1(43):139, 2001.

[34] Zhi Zhang, Yueqiang Cheng, Surya Nepal, Dongxi Liu, Qingni Shen, and Fethi Rabhi. KASR: A reliable and practical approach to attack surface reduction of commodity os kernels. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 691–710. Springer, 2018.

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

[36] Hsuan-Chi Kuo, Akshith Gunasekaran, Yeongjin Jang, Sibin Mohan, Rakesh B Bobba, David Lie, and Jesse Walker. Multik: A framework for orchestrating multiple specialized kernels. *arXiv preprint arXiv:1903.06889*, 2019.

[37] Muhammad Abubakar, Adil Ahmad, Pedro Fonseca, and Dongyan Xu. SHARD: Fine-Grained Kernel Specialization with Context-Aware Hardening. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[38] Ruowen Wang, Ahmed M Azab, William Enck, Ninghui Li, Peng Ning, Xun Chen, Wenbo Shen, and Yueqiang Cheng. SPOKE: Scalable Knowledge Collection and Attack Surface Analysis of Access Control Policy for Security Enhanced Android. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 612–624, 2017.

[39] Shankara Pailoor, Xinyu Wang, Hovav Shacham, and Isil Dillig. Automated policy synthesis for system call sandboxing. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–26, 2020.

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

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

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

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

[44] Darren Mutz, Fredrik Valeur, Giovanni Vigna, and Christopher Kruegel. Anomalous system call detection. *ACM Transactions on Information and System Security (TISSEC)*, 9(1):61–93, 2006.

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

[46] Federico Maggi, Matteo Matteucci, and Stefano Zanero. Detecting intrusions through system call sequence and argument analysis. *IEEE Transactions on Dependable and Secure Computing*, 2010.

[47] Raymond Canzanese, Spiros Mancoridis, and Moshe Kam. System call-based detection of malicious processes. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 119–124. IEEE, 2015.

[48] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. Temporal system call specialization for attack surface reduction. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1749–1766, 2020.

[49] Anup K Ghosh, Aaron Schwartzbard, and Michael Schatz. Learning program behavior profiles for intrusion detection. In *Workshop on Intrusion Detection and Network Monitoring*, 1999.

[50] Liangyi Gong, Zhenhua Li, Feng Qian, Zifan Zhang, Qi Alfred Chen, Zhiyun Qian, Hao Lin, and Yunhao Liu. Experiences of landing machine learning onto market-scale mobile malware detection. In *Proc. ACM EuroSys*, 2020.

[51] Brian Kondracki, Babak Amin Azad, Najmeh Miramirkhani, and Nick Nikiforakis. The droid is in the details: Environment-aware evasion of android sandboxes. In *Proc. Network and Distributed Systems Security Symposium (NDSS)*, 2022.