MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation

Shankara Pailoor, Andrew Aday, and Suman Jana
Columbia University

Abstract

OS fuzzers primarily test the system-call interface between the OS kernel and user-level applications for security vulnerabilities. The effectiveness of all existing evolutionary OS fuzzers depends heavily on the quality and diversity of their seed system call sequences. However, generating good seeds for OS fuzzing is a hard problem as the behavior of each system call depends heavily on the OS kernel state created by the previously executed system calls. Therefore, popular evolutionary OS fuzzers often rely on hand-coded rules for generating valid seed sequences of system calls that can bootstrap the fuzzing process. Unfortunately, this approach severely restricts the diversity of the seed system call sequences and therefore limits the effectiveness of the fuzzers.

In this paper, we develop MoonShine, a novel strategy for distilling seeds for OS fuzzers from system call traces of real-world programs while still preserving the dependencies across the system calls. MoonShine leverages light-weight static analysis for efficiently detecting dependencies across different system calls.

We designed and implemented MoonShine as an extension to Syzkaller, a state-of-the-art evolutionary fuzzer for the Linux kernel. Starting from traces containing 2.8 million system calls gathered from 3,220 real-world programs, MoonShine distilled down to just over 14,000 calls while preserving 86% of the original code coverage. Using these distilled seed system call sequences, MoonShine was able to improve Syzkaller's achieved code coverage for the Linux kernel by 13% on average. MoonShine also found 17 new vulnerabilities in the Linux kernel that were not found by Syzkaller.

1 Introduction

Security vulnerabilities like buffer overflow and use-after-free inside operating system (OS) kernels are particularly dangerous as they might allow an attacker to completely compromise a target system. OS fuzzing is a popular technique for automatically discovering and fixing such critical security vulnerabilities. Most OS fuzzers focus primarily on testing the system-call interface as it is one of the main points of interaction between the OS kernel and user-level programs. Moreover, any bug in system call implementations might allow an unprivileged user-level process to completely compromise the system.

OS fuzzers usually start with a set of *synthetic seed programs*, i.e., a sequence of system calls, and iteratively mutate their arguments/orderings using evolutionary guidance to maximize the achieved code coverage. It is well-known that the performance of evolutionary fuzzers depend critically on the quality and diversity of their seeds [31, 39]. Ideally, the synthetic seed programs for OS fuzzers should each contain a small number of system calls that exercise diverse functionality in the OS kernel.

However, the behavior of each system call heavily depends on the shared kernel state created by the previous system calls, and any system call invoked by the seed programs without the correct kernel state will only trigger the shallow error handling code without reaching the core logic. Therefore, to reach deeper into a system call logic, the corresponding seed program must correctly set up the kernel state as expected by the system call. As user programs can only read/write kernel state through other system calls, essentially the seed programs must identify the dependent system calls and invoke them in a certain system-call-specific order. For example, a seed program using the `read` system call must ensure that the input file descriptor is already in an "opened" state with read permissions using the `open` system call.

Existing OS fuzzers [11, 37] rely on thousands of hand-coded rules to capture these dependencies and use them to generate synthetic seed programs. However, this approach requires significant manual work and does not scale well to achieve high code coverage. A promising alternative is to gather system call traces from diverse existing programs and use them to generate synthetic seed programs. This is because real programs are *required* to satisfy these dependencies in order to function correctly.

However, the system call traces of real programs are large and often repetitive, e.g., executing calls in a loop. Therefore, they are not suitable for direct use by OS fuzzers as they will significantly slow down the efficiency (i.e., execution rate) of the fuzzers. The system call traces must be distilled while maintaining the correct dependencies between the system calls as mentioned earlier to ensure that their achieved code coverage does not

go down significantly after distillation. We call this process *seed distillation* for OS fuzzers. This is a hard problem as any simple strategy that selects the system calls individually without considering their dependencies is unlikely to improve coverage of the fuzzing process. For example, we find that randomly selecting system calls from existing program traces do not result in any coverage improvement over hand-coded rules (see Section 5.4 for more details).

In this paper, we address the aforementioned seed distillation problem by designing and implementing MoonShine, a framework that automatically generates seed programs for OS fuzzers by collecting and distilling system call traces from existing programs. It distills system call traces while still maintaining the dependencies across the system calls to maximize coverage. MoonShine first executes a set a real-world programs and captures their system call traces along with the coverage achieved by each call. Next, it greedily selects the calls that contribute the most new coverage and for each such call, identifies all its dependencies using lightweight static analysis and groups them into seed programs.

We demonstrate that MoonShine is able to distill a trace consisting of a total of 2.8 million system calls gathered from 3,220 real programs down to just over 14,000 calls while still maintaining 86% of their original coverage over the Linux kernel. We also demonstrate that our distilled seeds help Syzkaller, a state-of-the-art system call fuzzer, to improve its coverage achieved for the Linux kernel by 13% over using manual rules for generating seeds. Finally, MoonShine's approach led to the discovery of 17 new vulnerabilities in Linux kernel, none of which were found by Syzkaller while using its manual rule-based seeds.

In summary, we make the following contributions:

- We introduce the concept of seed distillation, i.e., distilling traces from real world programs while maintaining both the system call dependencies and achieved code coverage as a means of improving OS fuzzers.
- We present an efficient seed distillation algorithm for OS fuzzers using lightweight static analysis.
- We designed and implemented our approach as part of MoonShine and demonstrated its effectiveness by integrating it with Syzkaller, a state-of-the-art OS fuzzer. MoonShine improved Syzkaller's test coverage for the Linux kernel by 13% and discovered 17 new previously-undisclosed vulnerabilities in the Linux kernel.

The rest of the paper is organized as follows. Section 2 provides an overview of our techniques along with a motivating example. Section 3 describes our methodology. We discuss the design and implementation of MoonShine in Section 4 and present the results of our evaluation in Section 5. Finally, we describe related work in Section 8 and conclude in Section 10.

2 Overview

2.1 Problem Description

Most existing OS fuzzers use thousands of hand-coded rules to generate seed system call sequences with valid dependencies. As such an approach is fundamentally unscalable, our goal in this paper is to design and implement a technique for automatically distilling system calls from traces of real existing programs while maintaining the corresponding dependencies. However, system call traces of existing programs can be arbitrarily large and repetitive, and as a result will significantly slow down the performance of an OS fuzzer. Therefore, in this paper, we focus on distilling a small number of system calls from the traces while maintaining their dependencies and preserving most of the coverage achieved by the complete traces.

Existing test case minimization strategies like afl-tmin [12] try to dynamically remove parts of an input while ensuring that coverage does not decrease. However, such strategies do not scale well to program traces containing even a modest number of system calls due to their complex dependencies. For example, consider the left-hand trace shown in Figure 1. A dynamic test minimization strategy similar to that of afl-tmin might take up to 256 iterations for finding the minimal distilled sequence of calls.

To avoid the issues described above, we use lightweight static analysis to identify the potential dependencies between system calls and apply a greedy strategy to distill the system calls (along with their dependencies) that contribute significantly towards the coverage achieved by the undistilled trace. Before describing our approach in detail, we define below two different types of dependencies that we must deal with during the distillation process.

Explicit Dependencies. We define a system call c_i to be explicitly dependent on another system call c_j if c_j produces a result that c_i uses as an input argument. For example, in Figure 1, the `open` call in line 2 is an explicit dependency of the `mmap` call in line 3 because `open` returns a file descriptor (3) that is used by `mmap` as its fourth argument. If `open` did not execute, then `mmap` would not return successfully, which means it would take a different execution path in the kernel.

Implicit Dependencies. A system call c_i is defined to be implicitly dependent on c_j if the execution of c_j affects the execution of c_i through some shared data struc-

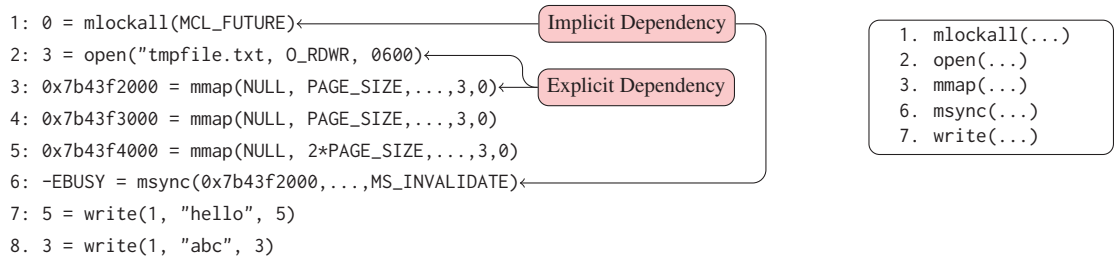

Figure 1: An example of seed distillation by MoonShine. On the left is an example trace before distillation and on the right are the calls MoonShine identified as contributing the most new coverage along with their dependencies. The line numbers on the right indicate their position in the original trace.

ture in the kernel, even though there is no overlap between c_j's output and c_i's input arguments. In Figure 1, the `mlockall` call is an implicit dependency of the `msync` call. The `mlockall` call instructs the kernel to lock all memory pages that are mapped into the process's address space to avoid swapping. When `msync` is called with the flag `MS_INVALIDATE` on an mmap'd page to invalidate all changes, `msync` fails with an `-EBUSY` error because the pages were locked in memory. In this case, the `mlockall` call affects the behavior of `msync` through the `vma->vm_flags` as shown in Figure 2 even though these calls do not share any arguments.

2.2 Motivating Example

MoonShine detects explicit and implicit dependencies by statically analyzing the system call traces and the kernel sources. We outline how MoonShine performs seed distillation by leveraging these dependencies below.

For distillation, MoonShine first identifies the calls that contribute the most unique code coverage. Let us assume that the `mmap`, `msync`, and `write` calls in lines 3, 6 and 7 respectively contribute most to the code coverage in this trace. For each such call, MoonShine uses static analysis on the trace to identify the explicit dependencies. For the `mmap`, MoonShine iterates over all its arguments and looks for any upstream calls in the trace where the argument was produced by a system call. In this case, the only argument that matches the result of an upstream call is the fourth argument: the file descriptor 3 matches the result of `open` in line 2. MoonShine applies the same procedure for the `msync` call and it finds that the first argument of `msync` matches the result of `mmap` in line 3 and so `mmap` is marked as an explicit dependency of `msync`. When MoonShine applies the same procedure to the `write` it finds that it does not have explicit dependencies.

Next, MoonShine uses static analysis on the kernel source code to identify any upstream calls that may be implicit dependencies of `msync`, `mmap`, and `write`. For `msync`, MoonShine discovers that `mlockall`'s execution can impact the coverage achieved by `msync`. It observes that `msync` checks the value of the `struct vma_struct->vma_flags` field and `mlockall` writes to the same field. Figure 2 shows the relevant code from the implementations of mmap and msync in the kernel. `mlockall` calls `mlock_fixup` which in turn sets the `vma_flags` field for every `struct vma_struct` in the calling process (line 7). In this case, `lock` on line 6 is true and `newflags` contains the bitflag `VM_LOCKED`. Without the `mlockall`, the `vm_flag` field would not be set, and `msync` would not return `-EBUSY`, as highlighted on line 5. MoonShine applies the same process to `mmap` and finds that `mlockall` is also an implicit dependency of `mmap`. In the case of the `write`, MoonShine again finds that it has no upstream dependencies.

Finally, MoonShine recursively identifies all the dependencies of the system calls that are identified in the last two steps described above. In this example, MoonShine finds that the `open` and `mlockall` calls have no dependencies in the trace. Therefore, MoonShine returns all the dependencies of `write`, `mmap` and `msync` as the distilled trace shown on the right in Figure 1.

3 Approach

We present MoonShine's core seed distillation logic in Algorithm 1. Starting from a list of system calls $\mathbb{S}$ gathered from the program traces, MoonShine sorts the system calls by their coverage from largest to smallest (line 8). For each call in the list, MoonShine captures both the explicit (line 11) and implicit dependencies (line 12). The dependencies, along with the system calls, are merged (line 14) so that their ordering in the distilled trace matches their ordering in the original trace. This grouping of distilled calls is added to our collection of seeds $\mathscr{S}$ (line 16) for OS fuzzing.

In Algorithm 1, we demonstrate that MoonShine constructs seeds from the calls that contribute the most new coverage and captures those calls' implicit and explicit dependencies. In this section we describe how Moon-

mlockall

```
1: int mlockall(...) {
2: ...
3: void mlock_fixup_lock(...)
4: {
5:     ...
6:     if (lock)
7:         vma->vm_flags = newflags;
8: }
```

msync

```
1: int msync(...)
2: {
3:     ...
4:     if ((flags & MS_INVALIDATE) &&
5:         (vma->vm_flags & VM_LOCKED) {
6:         error = -EBUSY;
7:     }
8: }
```

Figure 2: This listing shows an implicit dependency between `msync` and `mlockall`. The conditional of `msync` on the right depends on the value of the `struct vma_struct` which is set by `mlockall` on the left.

Algorithm 1 MoonShine's seed distillation algorithm for distilling trace S

```
1:  procedure SEEDSELECTION(𝕊)
2:      𝒮 = ∅
3:      ℂ = ∅
4:      i = 1
5:      for s ∈ 𝕊 do
6:          cov[i] = Coverage(s)
7:          i = i + 1
8:      sort(cov) // Sort calls by coverage
9:      for i = 1 → |S| do
10:         if cov[i] \ ℂ ≠ ∅ then
11:             expl_deps = GET_EXPLICIT(cov[i])
12:             impl_deps = GET_IMPLICIT(cov[i])
13:             deps = expl_deps ∪ impl_deps
14:             seed = MERGE(deps ∪ cov[i])
15:             ℂ∪ = cov[i]
16:             𝒮 = 𝒮 ∪ seed
17:     return 𝒮
```

Algorithm 2 Pseudocode for capturing explicit and implicit dependencies.

```
1:  procedure GET_EXPLICIT(c)
2:      deps = ∅
3:      𝒯 = TRACE_OF(T)
4:      DG = build_dependency_graph(𝒯)
5:      for arg in c.args do
6:          expl_deps = DG.neighbors
7:          for expl_dep in expl_deps do
8:              deps ∪ = GET_IMPLICIT(expl_dep)
9:              deps ∪ = {expl_dep}
10:     return deps
11: procedure GET_IMPLICIT(c)
12:     impl_deps = ∅
13:     for uc in upstream_calls(c) do
14:         if uc.WRITE_deps ∩ c.READ_deps then
15:             impl_deps ∪ = GET_EXPLICIT(uc)
16:             impl_deps ∪ = {uc}
17:     return deps
```

Shine captures those dependencies.

Explicit Dependencies. For each trace, MoonShine builds a dependency graph that consists of two types of nodes: results and arguments. Result nodes correspond to values returned by system calls. The result nodes store the following information: 1) value returned, 2) return type (pointer, int, or semantic) and 3) the call in the trace which produced the result. Argument nodes similarly store the value of the argument, the type, and the call to which the argument belongs. An edge from argument node a to result node r indicates that a's value relies on the call which produced r. MoonShine builds the graph as it parses the trace. For the returned value of each call, it constructs the corresponding result node and adds it to the graph. Afterwards, it places the result node in a result map that is indexed using the composite key of (type, value). For each argument in a call, MoonShine checks the result cache for an entry. A hit indicates the existence of at least one system call whose result has the same type and value as the current argument. MoonShine iterates over all the result nodes stored in the map for the specific type and value and adds one edge from the argument node to each result node in the graph.

Once the argument dependency graph is constructed, MoonShine identifies explicit dependencies for a given call by enumerating the call's list of arguments and for each argument MoonShine visits the corresponding argument node in the dependency graph. For every edge from the argument node to a result node, MoonShine marks the calls that produced the result node as an explicit dependency. After traversing the entire list, MoonShine returns all calls marked as explicit dependencies.

Implicit Dependencies. In order for the coverage achieved by a system call c_i to be affected by the prior execution of system call c_j, c_j's execution must influence the evaluation of a conditional in c_i's execution. This is because the only values that can be used to evaluate a conditional are those that are passed as arguments or those existing in the kernel. Therefore, if a call c_i is an implicit dependency of call c_j then c_j must have a condi-

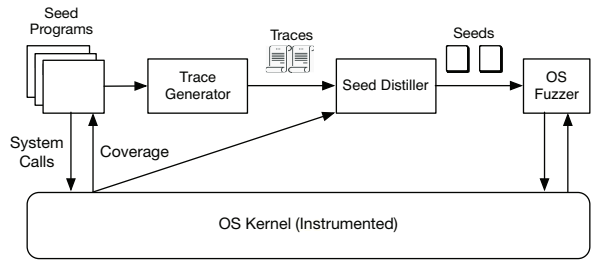

Figure 3: MoonShine workflow

tional in its control flow which depends on a global value v that is modified by c_i.

This gives rise to the following definitions. A global variable v is a **read dependency** of a system call c if c reads v in a conditional. Similarly, a global variable v is a **write dependency** of a system call c if c ever writes to v. As such, a call c_a is an implicit dependency of c_b if the intersection of c_a's write dependencies and c_b's read dependencies is nonempty.

MoonShine is able to identify the collection of read and write dependencies by performing control flow analysis on the target kernel. For a given system call, the flow analysis starts at the function definition. At each conditional, MoonShine checks all components of the corresponding expression and records all global variables read. If MoonShine encounters an assignment expression or unary assignment expression containing a global variable, it marks that global variable as a write dependency.

Note that for a given trace this approach may overestimate or underestimate the number of implicit dependencies for a given call. It may overestimate because the condition for which the global variable is a read dependency may only be taken for specific values. Calls that write to that field may not necessarily write the required values of the conditional. This approach can underestimate the dependencies if the variable is aliased and that aliased variable is used in the conditional instead. This method can be further refined through "fine-grained" data flow analysis, but this comes at the cost of efficiency during distillation.

The pseudocode for these routines is described in Algorithm 2. Note that the implicit and explicit routines recursively call each other. This is because every upstream dependency must have its dependencies captured as well. This recursive procedure will always terminate because in each iteration the target call gets closer to the beginning of the trace.

4 Implementation

We present MoonShine's workflow in Figure 3. MoonShine consists of two components: Trace Generation and Seed Selection. During trace generation, MoonShine executes our seed programs on a kernel instrumented to record coverage and captures their system call traces. This collection of traces is passed to the Seed Distiller which applies our distillation algorithm to extract seeds for the target fuzzer.

Kernel Instrumentation. In order to perform distillation, MoonShine needs to know the coverage reached by each system call inside the kernel during its execution. In general this can be achieved at compile time or through binary instrumentation. In our prototype we compile Linux with the flag `CONFIG_KCOV` [38] which instruments the kernel with gcc's sanitizer coverage. Linux allows privileged user level programs to recover the coverage they achieved through the debugfs file /sys/kernel/fs/debug/kcov. During fuzzing we combine multiple other gcc sanitizers to detect bugs, namely Kernel Address Sanitizer (KASAN) [18] and Kernel UndefinedBehaviorSanitizer (UBSAN) [14]. We also enable kernel-specific detectors like the Lock dependency tracker for deadlocks and KMEMLEAK [5] for memory leaks.

Tracer. We implement our tracer by adapting and extending Strace [13], a popular system call tracer. We extended Strace because it captured system call names, arguments, and return values out-of-the-box. Furthermore, Strace can track calls across `fork` and `exec` which is useful because many programs are executed by using scripts and if we are unable to capture traces across these calls then it limits our ability to scalably capture traces. Our extension adds a total of 455 lines of code across 3 files. This feature is disabled by default but can be enabled by running `Strace` with the `-k` flag. We plan to submit a patch of our changes to the Strace maintainers.

Multiprocess Traces. If a trace consists of multiple processes, MoonShine first constructs a process tree. Every node in the tree stores the system call traces for that specific process. An edge from node A to node B indicates that B is a child of A. MoonShine determine this relationship by examining the return value of the `clone` system call. If process A calls `clone` and the result is $B > 0$ then we know A is a parent of B. Each edge also stores the position of the last call in A's trace before B was created, and this is important because some resources produced by A can be accessed by B, e.g. file descriptors or memory mappings. MoonShine builds a dependency graph for each node in the tree in DFS order. Each node in the dependency graph also stores the position of the call in that processes trace. When computing the explicit dependencies for a call in a trace MoonShine first checks the local dependency graph. If that value is

not in the cache then it traverses up the process tree and checks each process argument graph. If there is a hit in the parent process, MoonShine checks to make sure that the value was stored in the cache prior to the `clone`. In this case, MoonShine will copy the call and its upstream dependencies into the child's trace.

Explicit Dependencies. There are three exceptions to our approach of capturing explicit dependencies. First, system call arguments may *themselves* return results e.g, `pipe`. In order to track this, MoonShine requires the aid of a template that identifies for a given system call, which argument has its values set by the kernel. With such a template, MoonShine will also store the value returned in the argument inside of its result cache. Second, memory allocation calls like `mmap` return a *range* of values. A system call may depend on a value inside the range but not on the value explicitly returned. MoonShine handles this by specifically tracking memory allocations made by `mmap` or `SYSTEM V` calls. As it parses the trace it makes a list of active mappings. If the value of a pointer argument falls within an active mapping, then MoonShine adds an edge from the argument to the call that produced that mapping. For any pointer values that do not fall within an active mapping, such as those on the stack or produced through `brk`, MoonShine tracks the memory required for all such arguments and adds a large `mmap` call at the beginning of the distilled trace to store their values. The final exception is when two seeds, currently placed in separate distilled programs, are found to be dependent on one another. In this case, MoonShine merges the two programs into one.

Implicit Dependencies. MoonShine's implicit dependency tracker is build on Smatch [16], a static analysis framework for C. Smatch allows users to register functions which are triggered on matching events while Smatch walks the program's AST. These hooks correspond to C expressions such as an Assignment Hook or Conditional Hook. MoonShine tracks read dependencies by registering a condition hook that checks if the conditional expression, or any of its subexpressions, contains a struct dereference. On a match, the hook notifies MoonShine which struct and field are the read dependency along with the line and function name, which MoonShine records.

MoonShine tracks write dependencies by registering a Unary Operator Hook and Assignment Hook. The unary operator hook notifies MoonShine every time a unary assignment operation is applied to a struct deference. The notification describes the corresponding struct name and field and MoonShine records the struct and field as a write dependency. Our assignment hook is nearly identical except it only checks the expression on the left side of the assignment. After running Smatch with our hooks, we generate a text file that is read by our distillation algorithm to identify potential implicit dependencies for every call.

5 Evaluation

In this section we evaluate the effectiveness of MoonShine both in terms of its ability to aid OS fuzzers in discovering new vulnerabilities, as well as in terms of its efficiency in gathering and distilling traces while preserving coverage. In particular, we assessed MoonShine's impact on the performance of Syzkaller, a state-of-the-art OS fuzzer targeting the Linux kernel, by distilling seeds constructed from traces of thousands of real programs. Our evaluation aims at answering the following research questions.

- **RQ1**: Can MoonShine discover new vulnerabilities? (Section 5.2)
- **RQ2**: Can MoonShine improve code coverage? (Section 5.3)
- **RQ3**: How effectively can MoonShine track dependencies? (Section 5.4)
- **RQ4**: How efficient is MoonShine? (Section 5.5)
- **RQ5**: Is distillation useful? (Section 5.6)

5.1 Evaluation Setup

Seed Programs. Since MoonShine's ability to track dependencies is limited to the calls within a single trace, we sought out seed programs whose functionality is *self-contained*, but also provides diverse coverage. We constructed seeds from 3220 programs from the following sources 1) Linux Testing Project (LTP) [7], 2) Linux Kernel selftests (kselftests) [6], 3) Open Posix Tests [8], 4) Glibc Testsuite [3].

The LTP testsuite is designed to test the Linux kernel for reliability, robustness and scalability and is curated by both kernel developers along with third party companies such as IBM, Cisco, and Fujitsu. Out of LTP's 460 system call tests we collected traces for 390. The testcases we avoided focused on system calls which Syzkaller does not support such as `execve`, `clone`, `cacheflush`, etc.

Kselftests is a testing suite contained within the Linux source tree that tests specific subsystems in the kernel. Like with our LTP traces, most of the kselftest traces were collected from the system call suite. Although this testsuite is significantly smaller than LTP we chose to collect from it because it is designed to test specific paths through the kernel. As such, we can expect each program to provide diverse coverage and be reproducible.

The OpenPosix test suite is designed to test the Posix 2001 API specifications for threads, semaphores, timers and message queues. We collected traces from the 1,630 message queue and timer tests.

The glibc test suite is used for functional and unit testing of glibc. The test suite includes regression tests against previously discovered bugs, and tests which exercise components of the C Standard Library such as processing ELF files, io, and networking calls. We collected the traces from 1,120 glibc tests.

OS Fuzzer. In the experiments we used Syzkaller as our target OS fuzzer. We chose Syzkaller as it is a state-of-the-art system call fuzzer, having found a large number of vulnerabilities, and is actively maintained. Furthermore, Syzkaller employs effective strategies to discover non-deterministic bugs, e.g., by occasionally executing calls from a given program on different threads. Syzkaller also combines many other existing bug finding mechanisms like fault injection to trigger bug inducing scenarios. Unless stated otherwise, we configured Syzkaller to run on Google Compute Engine (GCE) with 2 fuzzing groups, each group containing 4 fuzzing processes. The Syzkaller manager ran on an Ubuntu 16.04 n1-standard-1 instance which contains 1vCPU and 3.75GB. Each fuzzing group ran on an n1-highcpu-4 machine consisting of 4vCPUs and 3.60GB of memory running our target kernel.

Distillation Algorithms. In this evaluation we compare MoonShine's distillation algorithm, termed **Moonshine(I+E)**, against two others. The first is a distillation algorithm which only captures the explicit dependencies, ignoring implicit dependencies, which we call **MoonShine(E)**. The second is a random distillation algorithm, called **RANDOM**, which tracks no dependencies at all. The RANDOM algorithm works by first selecting all system calls in a trace that contributed the most coverage increase, and assigning each to its own synthetic program. Then it randomly selects system calls from across all program traces, distributing them evenly across the synthetic programs, until it has pulled as many system calls as Moonshine(I+E).

Lastly, we use the term **default Syzkaller** to describe Syzkaller fuzzing without any seeds, using only it's hard-coded rules to generate input programs.

5.2 Can MoonShine discover new vulnerabilities? (RQ1)

Table 1 shows the vulnerabilities in the Linux kernel that were discovered using MoonShine. Each vulnerability was triggered during a fuzzing experiment that lasted 24 hours. Each experiment consisted of the following steps. First, we generate two sets of distilled seeds using Moonshine(I+E) and MoonShine(E) on traces gathered from all our seed programs. For each set of seeds, we fuzz the latest kernel release candidate for 24 hours 3 times each and do the same using the default Syzkaller. For a vulnerability to be considered as caused by one set of seeds, it must be triggered in at least two of the three experiments and not by default Syzkaller. During each experiment, we restricted Syzkaller to only fuzz the calls contained in our traces to more accurately track the impact of our seeds. We note that default Syzkaller was unable to find any vulnerabilities during these experiments but when using seeds generated by MoonShine it found 17.

Vulnerabilities Results. Of the 17 new vulnerabilities we discovered, 10 of them were only discovered when using seeds generated by Moonshine(I+E) and the average age of each was over 9 months. Two of the vulnerabilities we found in `fs/iomap.c` and `iomap_dio_rw` were over 4 years old. We also note that each of the bugs discovered using Moonshine(I+E) alone were concurrency bugs that were triggered by Syzkaller scheduling calls on different threads. We also note that our bugs were found in core subsystems of the kernel, namely `VFS` and `net/core`. We have reported all vulnerabilities to the appropriate maintainers and 9 have already been fixed.

Result 1: MoonShine found 17 new vulnerabilities that default Syzkaller cannot find out of which 10 vulnerabilities can only be found using implicit dependency distillation.

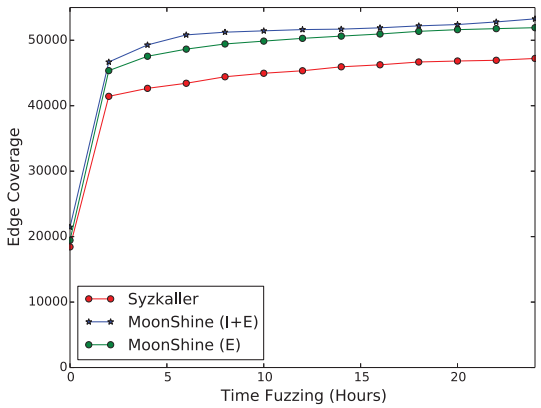

Figure 4: Coverage achieved using Moonshine(I+E) and MoonShine(E) and default Syzkaller in 24 hours of fuzzing. Seed traces were obtained from the LTP, Kselftest, Glibc, and Posix sources.

Subsystem	Module	Operation	Impact	Version Introduced	Distill. Method
BPF	bpf/devmap.c	dev_map_alloc()	Illegal allocation size	4.0	(I+E) & (E)
BTRFS	fs/btrfs/file.c	btrfs_fallocate()	Assert Failure	4.14	(I+E)
Ext4	fs/fs-writeback.c	move_expired_inodes()	Use After Free	4.6	(I+E)
JFS	fs/jfs/xattr.c	__jfs_setxattr()	Memory Corruption	2.6	(I+E) & (E)
Network	net/ipv4/inet_connection_sock.c	inet_child_forget()	Use after Free	4.4	(I+E)
Network	net/core/stream.c	sk_kill_stream_queues()	Memory Corruption	4.4	(I+E)
Network	net/core/dst.c	dst_release()	NULL Pointer Deref	4.15-rc8	(I+E)
Network	net/netfilter/nf_conntrack_core.c	init_conntrack()	Memory Leak	4.6	(I+E)
Network	net/nfc/nfc.h	nfc_device_iter_exit()	NULL Pointer Deref	4.17-rc4	(I+E)
Network	net/socket.c	socket_setattr()	NULL Pointer Deref	4.10	(I+E) & (E)
Posix-timers	kernel/time/posix-cpu-timers.c	posix_cpu_timer_set()	Integer Overflow	4.4	(I+E) & (E)
Reiserfs	fs/reiserfs/inode.c, fs/reiserfs/ioctl.c, fs/direct-io.c	Multiple	Deadlock	4.10	(I+E)
TTY	tty/serial/8250/8250_port.c	serial8250_console_putchar()	Kernel Hangs Indefinitely	4.14-rc4	(I+E)
VFS	fs/iomap.c	iomap_dio_rw()	Data Corruption	3.10	(I+E) & (E)
VFS	lib/iov_iter.c	iov_iter_pipe()	Data Corruption	3.10	(I+E) & (E)
VFS	fs/pipe.c	pipe_set_size()	Integer Overflow	4.9	(I+E) & (E)
VFS	inotify_fsnotify.c	inotify_handle_event()	Memory Corruption	3.14	(I+E)

Table 1: List of previously unknown vulnerabilities found by MoonShine. The rightmost, **Distill. Method** column reports which distillation methods produced the seeds that led Syzkaller to find said vulnerability. (I+E) is shorthand for Moonshine(I+E), and (E) for MoonShine(E). No vulnerabilities were found by the undistilled traces or default Syzkaller.

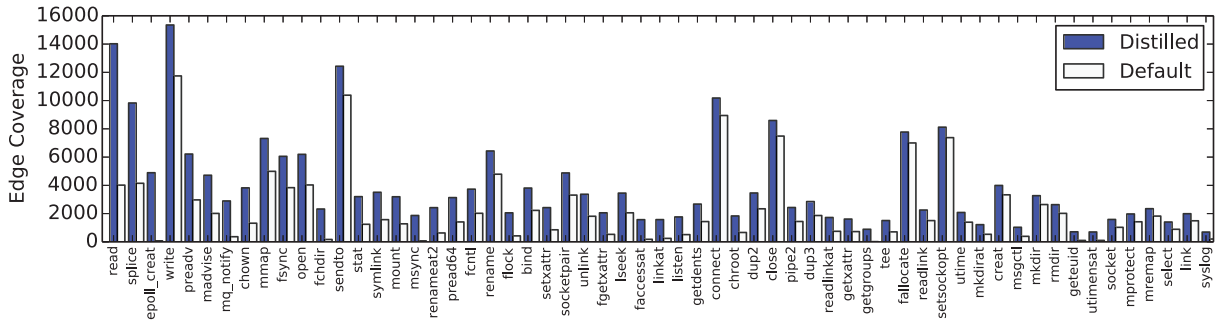

Figure 5: Coverage breakdown by system call after twenty four hours of fuzzing. The dark blue bars are for Moonshine(I+E) and white bars are for default Syzkaller.

5.3 Can MoonShine improve code coverage? (RQ2)

Figure 4 shows the coverage achieved by our Moonshine(I+E) and MoonShine(E) algorithms compared to Syzkaller using only its manual rules over 24 hours of fuzzing. The seeds used in this experiment are generated from all our seed programs described in Section 5.1. For a fair comparison, we restrict Syzkaller to only fuzz the system calls that were present in our traces.

Overall Coverage Results. For edge coverage, Moonshine(I+E) covered 53,270 unique basic blocks, MoonShine(E) covered 51,920 and default Syzkaller covered 47,320. This shows Syzkaller's coverage improves noticeably when it starts with either of MoonShine's generated seeds; however, when seeded with programs that have been distilled with both explicit *and* implicit dependencies, Syzkaller achieves 13% coverage improvement compared to the 9% when using explicit dependencies alone.

Breakdown By System Call. Figure 5 shows the breakdown of the coverage achieved by Moonshine(I+E) compared to default Syzkaller by system call. The height of each bar represents the union of all unique basic

blocks hit by that system call across all fuzzer programs (both seed and generated) over 24 hours of fuzzing. We see that the system calls where Moonshine(I+E) outperformed default Syzkaller were among *standard* system calls such as `read`, `write`, `fsync` and `mmap`. This fact that Moonshine(I+E) noticeably outperformed Syzkaller on these standard system calls suggests that Syzkaller's hard coded rules are insufficient to capture dependencies for common calls.

Coverage and Bugs Found. Although 10 out of 17 bugs found were concurrency related we observed that all our concurrency bugs were found in the file and networking subsystems. Similarly, the calls which produced the most new coverage under our distilled seeds were also file or networking related, for example `fsync` and `sockpair`. This correlation is not arbitrary. Since Syzkaller is a coverage-guided, evolutionary fuzzer, it will continually stress the programs and system calls which are returning the most new coverage. The test suites we used for source programs contain programs which especially exercise functionality in the networking and filesystem kernel subsystems. Of the 17 bugs found by MoonShine, 6 were from the network subsystem, and 8 from file systems. These findings imply that the composition of seed programs is able to influence Syzkaller to focus on fuzzing particular regions of the kernel it otherwise would not, and in extension discover bugs in these regions.

Result 2: MoonShine achieves 13% higher edge coverage than default Syzkaller

5.4 How effectively can MoonShine distill traces? (RQ3)

Tracking Dependencies. To evaluate how effectively MoonShine can track dependencies, we first measured the coverage achieved by our seed programs during trace generation. Afterwards, we distilled these traces using Moonshine(I+E) and MoonShine(E) and measured the coverage achieved by Syzkaller due to these seeds alone, i.e. with mutation and seed generation disabled. We then compared the intersection of the coverage achieved by our traces and the coverage achieved by Syzkaller. Table 2 shows the result of this experiment as we expanded our seed program sources.

The left column indicates the seed programs used in the experiment. As we expanded the number of seed programs, Syzkaller recovered 86.8% and 78.6% of the original trace coverage using seeds generated by Moonshine(I+E) and MoonShine(E). To understand the impact of tracking dependencies, we repeated this experiment using seeds generated by RANDOM. As we see in Column 3, the coverage recovered is at most 23%, nearly four times worse than when using seeds generated by Moonshine(I+E) and MoonShine(E).

To understand the impact of the coverage we could not recover, we repeated our experiments but allowed Syzkaller to mutate and generate seeds. After 30 minutes, we recompared the coverage intersection. The results are summarized in Table 3. When using Moonshine(I+E) and MoonShine(E), Syzkaller can recover 95% and 91.6% of the original traces but when using RANDOM it achieves minimal improvement over default Syzkaller. This suggests that capturing dependencies is crucial to improving Syzkaller's performance and that MoonShine is able to do so effectively.

Result 3: MoonShine distills 3220 traces consisting of 2.9 million calls into seeds totaling 16,442 calls that preserve 86% of trace coverage.

5.5 How efficient is MoonShine? (RQ4)

To evaluate the efficiency of MoonShine, we measured the execution time of each of MoonShine components across our different sources. These results are summarized in Table 4. The last row shows the time required to process all our sources at once through MoonShine's workflow.

Trace Generation. Prior to benchmarking our components, we preloaded all seed programs on a custom Google Cloud image running linux-4.13-rc7 compiled with `kcov`. During trace generation, we launched 4 n1-standard-2 machines and captured the traces in parallel. Our results show that our modifications to Strace result in a 250% slowdown during trace generation uniformly across sources. However, this is to be expected because after each system call we must capture the coverage recorded by `kcov` and write it to a file. Furthermore, the `kcov` api does not deduplicate the edge coverage achieved by a call during its execution. We found that without deduplication, the average size of our traces were 33MB. By deduplicating the instructions during trace generation we are able to reduce the average trace size from 33MB to 102KB.

Distillation. Our results for distillation show that the time required to distill a source was proportional to the size of the source. As Table 4 demonstrates, it took only 18 minutes to distill 3220 traces that contained over 2.9 million calls. We also found that over 90% of the execution time in distillation was spent reading the traces. The time required to build the dependency graph and track implicit dependencies was only 30 seconds. This suggests that MoonShine is able to distill efficiently.

Source	Coverage				Number of Distilled Calls			
	Traced	**RANDOM**	**(E)**	**(I+E)**	**Traced**	**RANDOM**	**(E)**	**(I+E)**
L+K	19,500	3,460 (17.7%)	13,320 (68.3%)	16,400 (84.1%)	283,836	12,712 (4.47%)	10,200 (3.6%)	12,712 (4.47%)
P+L+K	23,381	5,532 (23.7%)	18,288 (78.2%)	21,432 (91.6%)	1,863,474	15,333 (0.82%)	11,455 (.61%)	15,333 (0.82%)
P+G+L+K	25,240	5,449 (21.6%)	19,840 (78.6%)	21,920 (86.8%)	2,953,402	16,442 (0.56%)	11,590 (.39%)	16,442 (0.56%)

Table 2: Seed source breakdown by distillation algorithm. The **Traced** columns report numbers from the original system call traces, prior to any distillation. **(I+E)** is short for Moonshine(I+E) and **(E)** for MoonShine(E). The numbers show the breakdown for our seed programs gathered from LTP (L), Posix Test Suites (L), Glibc Tests (G), Kselftests (K).

Distillation Method	Coverage Recovered	Percentage
I+E	24,230	95.0%
E	23,140	91.6%
RANDOM	19,120	75.7%
Default	18,200	72.1%

Table 3: Coverage recovered from original traces after 30 minutes of fuzzing. I+E refers to Moonshine(I+E) strategy and E refers to MoonShine(E).

Source	Trace w/ Coverage (mins)	Trace w/o Coverage (mins)	Distillation (mins)
L+K	8.5	3.8	4.3
G	28.4	13.3	8.5
P	20.4	7.7	10.5
Combined	**61.3**	**25.2**	**18.3**

Table 4: Breakdown of MoonShine performance across three seed program groups. The first is a combined LTP (L) + Kselftests(K), followed by Glibc (G) and finally Posix Test Suite (P).

Result 4: MoonShine collects and distills 110 gigabytes of raw program traces in under 80 minutes.

5.6 Is distillation useful? (RQ5)

We now evaluate our claim that without distillation the performance of the fuzzer will decrease significantly. We construct 5 different sets of seeds where the average number of calls for each seed increases by 146 but the number of seeds stay fixed at 500. We then instrument Syzkaller to record any mutations it performs on its programs. Each of our sets of seeds is used by Syzkaller as it fuzzes Linux 4.14-rc4 for 2 hours. Figure 6 shows the number of mutations it performs over the two hours. We observe that as the average length increases, the number of mutations decrease significantly. When using seeds whose average call length is 730, Syzkaller performed less than 100 mutations in one hour, which is prohibitively slow.

Distillation Method	Mutations/sec
Default	335
MoonShine(E)	305
Moonshine(I+E)	296
Undistilled	160

Table 5: Syzkaller's executions/sec measured after 2 hours of fuzzing across seeds generated from our different distillation algorithms. Our seed programs included LTP, Kselftests, Glibc teststuites, and Posix testsuites

We now assess the impact that MoonShine's seeds have on Syzkaller's overall performance. We measured the mutations per second achieved by Syzkaller throughout its 2 hour execution when using seeds generated by Moonshine(I+E), MoonShine(E), and with undistilled seeds. The results are summarized in Table 5. Syzkaller's baseline performance was 335 mutations per second. When using seeds generated by MoonShine(E) and Moonshine(I+E), the performance only decreased 10%. However, when Syzkaller used undistilled seeds, its mutation rate decreased by 53%.

Result 5: Running Syzkaller with undistilled seeds slows the mutation rate by 53%. Running Syzkaller on distilled seeds only reduces the mutation rate to 88.4% of what is achieved by default Syzkaller.

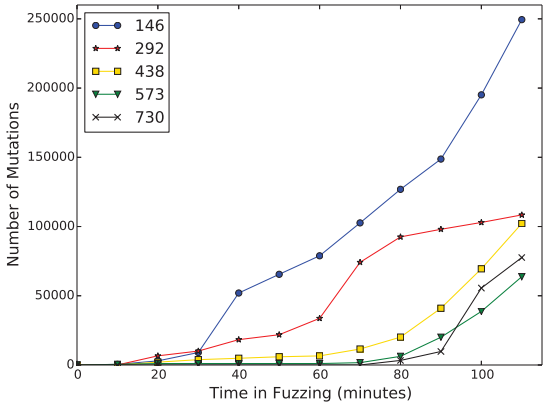

Figure 6: A comparison of Syzkaller's total mutations achieved in 2 hours of fuzzing while varying average seed program length. As program length increased, the number of mutations decreased in that timespan.

6 Case Studies of Bugs

In this section we describe two select bugs discovered by MoonShine during our experiments.

6.1 `inotify` Buffer Overflow

Description. Our first bug is a buffer overflow in the `inotify_handle_event()` module within the `inotify` subsystem. The inotify API enables users to track actions within the filesystem such as file creation, deletion, renaming, etc,. On a matching action, the kernel calls `inotify_handle_event()` to generate an event for the user, which has the following structure:

```
struct inotify_event_info {
    struct fsnotify_event *fse;
    int wd;
    u32 sync_cookie;
    int name_len;
    char name[]; /* optional field */
}
```

When generating a file-related event, `inotify_handle_event()` determines the amount of memory to allocate by first checking the length of the filename. After allocating memory, `inotify_handle_event()` calls `strcpy()` to copy the filename. However, if the filename length increases after determining the amount of memory to allocate but before the `strcpy()`, it will cause a buffer overflow. This can happen if another task calls `rename()` in that window. This scenario is detailed in Figure 7. In the top window, Thread 1 executes `inotify_handle_event` and if the event corresponds to a filename then it will call `strlen(dentry->d_name.name)`. After computing `alloc_len`, Thread 2 calls `rename` which performs a `memcpy` to change `dentry->d_name.name`. When Thread 1 resumes, `dentry->d_name.name` is different so the subsequent `strcpy` will overflow the struct if the size of the name has increased.

After 4.5 hours of fuzzing with seeds distilled using Moonshine(I+E), Syzkaller reported a *KASAN: slab out of bounds in strcpy* crash in `inotify_handle_event()`. The program that triggered the bug is listed in Figure 8. Lines 2 and 3 initialize an `inotify` instance to watch the current directory for all events. Line 5 creates a file named "short" and line 6 closes it. In line 7, the file is renamed to the longer name "long_name." The reason Syzkaller triggered this bug is because it will randomly schedule calls on different threads. In this case, the `rename` and `close` were run in parallel.

How Distilled Seeds Helped. The program in Figure 8 is from the `inotify02` testcase in LTP. The goal of the test case was to test the close, create, and rename events to ensure correct semantic behavior. When using only its manual rules, Syzkaller never generated the relevant sequence of calls for this bug to trigger. This is because its manual rules are weighted to select calls that share semantic types. In this case, the `rename`, `close` and `inotify_add_watch` did not share semantic types, but MoonShine's distillation algorithm could detect that each of these calls contributed new coverage as during their control paths each triggered an `inotify` event. Furthermore, MoonShine observed `inotify_add_watch` is an implicit dependency of both `rename` and `close` so the calls were merged into one program.

6.2 Integer Overflow in `fcntl`

The `pipe` system call creates an undirected data channel that allows communication between two processes. By default, the size of a pipe is the same as the system limit which is typically 4096 bytes. The size can be increased by calling `fcntl` with the command `F_SETPIPE_SZ`. However, calling this command with size 0 causes an `unsigned long long` overflow. Figure 9 shows the relevant excerpts from the call stack.

The root cause of the error happens in line 12. Since `size` is 0, `nr_pages` is also set to 0 which means that `fls_long(-1)` returns 64, resulting in the undefined expression `(1UL << 64)`.

How Distilled Seeds Helped. Our seed program fcntl30.c (from the LTP testsuite) called `fcntl` with `F_SETPIPE_SZ`. Figure 10 shows the relevant excerpt where the test iteratively changes the pipe size starting from the default size of 4096; however, during fuzzing, Syzkaller changed the size to 0. Default Syzkaller was unable to detect the bug because it is unable to understand that the command `F_SETPIPE_SZ` is meant to take

Thread 1: fs/notify/inotify/inotify_fsnotify.c

```
int inotify_handle_event()
{
    struct inotify_event_info *event;
    int len = 0;
    int alloc_len = sizeof(struct
        inotify_event_info);
    if (dentry->d_name.name) {
      len = strlen(dentry->d_name.name);
      alloc_len += len + 1;
    }
    /* Interrupted by Thread 2 */
```

Thread 2: fs/dcache.c

```
static void copy_name()
{
  memcpy(dentry->d_iname, target->d_name.name,
      target->d_name.len + 1);
  dentry->d_name.name = dentry->d_iname;
}
```

Thread 1 (continued)

```
    /* Execution Resumed */
    event = kmalloc(alloc_len, GFP_KERNEL);
    event->name_len = len;
    if (len)
      strcpy(event->name, dentry->d_name.name);
}
```

Figure 7: inotify_handle_event() bug in fs/notify/fsnotify.c. After Thread 1 computes alloc_len, Thread 2 increases the length of filename by copying a larger string to dentry->d_name.name, causing the overflow in strcpy.

```
1:  mmap(...)
2:  r0 = inotify_init()
3:  r1 = inotify_add_watch(r0,
        &(0x7f0000000000)="2e", 0xfff)
4:  chmod(&(0x7f0000001000)="2e", 0x1ed)
5:  r2 = creat(&(0x7f0000002000)="short",
        0x1ed)
6:  close(r2)
7:  rename(&(0x7f000000a000)="short",
        &(0x7f0000006000-0xa)="long_name")
8:  close(r0)
```

Figure 8: Syzkaller program that caused the bug. We have increased readability by truncating arguments and changing the filenames from hex strings to "long_name" and "short." Critically, "long_name" is longer than "short."

a file descriptor corresponding to pipe. When executing the command, Syzkaller randomly chooses from the collection of previously opened file descriptors so in order to trigger this bug it must select both fcntl with the command F_SETPIPE_SZ and ensure that pipe has already been executed. Whereas for the seed programs, the application already knows that the fcntl command should be associated with pipe so those two commands are already in the same program.

```
1:  long pipe_fcntl(...) {
2:     ...
3:     case F_SETPIPE_SZ:
4:         pipe_set_size(pipe, arg); //arg = 0
5:     ...
6:  long pipe_set_size(pipe, 0) {
7:     ...
8:         round_pipe_size(0);
9:     ...
10: unsigned int round_pipe_size(0) {
11:    ...
12:    nr_pages = (size + PAGE_SIZE - 1) >>
           PAGE_SHIFT; // = 0UL
13:    return (1UL << fls_long(0-1)) <<
           PAGE_SHIFT; //fls_long(-1) returns
           64
14: ...
```

Figure 9: fcntl undefined behavior when called with command F_SETPIPE_SZ and size of 0.

```
1:  int main(...)
2:  {
3:    int pipe_fds[2], test_fd;
4:    ...
5:    for (lc = 0; TEST_LOOPING(lc); lc++) {
6:      pipe(pipe_fds);
7:      test_fd = pipe_fds[1];
8:
9:      TEST(fcntl(test_fd, F_GETPIPE_SZ));
10:
11:     orig_pipe_size = TEST_RETURN;
12:     TEST(fcntl(test_fd, F_SETPIPE_SZ,
            new_pipe_size));
13:     ...
14:   }
15:   ...
16: }
```

Figure 10: Relevant excerpt from fcntl30.c. Traces from this program were distilled to form the fcntl pipe bug.

7 Discussion

We have demonstrated that trace distillation can improve kernel security by discovering new vulnerabilities efficiently. In this section, we describe some of the limitations of our current prototype implementation and some future directions that can potentially minimize these issues.

7.1 Limitations

Lack of Inter-Thread Dependency Tracking. MoonShine's dependency tracking algorithm assumes that all

dependencies of a call are produced by the same thread or a parent process. However, if a call depends on a resource produced by a parallel thread or process, then the current implementation of MoonShine cannot track the dependency. While the programs producing the traces used in this paper contained very few such inter-process/thread dependencies, more complex programs like databases or Web servers may have such dependencies as their processes/threads often share sockets and memory regions. Developing a tracking mechanism for such inter-thread/inter-process dependencies will be an interesting area for future work.

False Positives from Static Analysis. MoonShine's static implicit dependency analysis may result in false positives, i.e., it may detect two system calls to have implicit dependencies where there are none. Note that these false positives do not affect the coverage achieved by the distilled corpus but might make the traces slightly larger than they need to be.

In our experiments, we observed that imprecise pointer analysis is a major source of false positives. If two system calls read and write from the same struct field, MoonShine cannot determine if the corresponding pointers refer to the same struct instance. For example, MoonShine identifies `mlock` as an implicit dependency of `munmap` because `struct vma` is a write dependency of `mlock` and a read dependency of `munmap`. However, the instances of `struct vma` are completely determined by the pointers passed in as the first argument to each call. If the first arguments to these calls are different, then the instances of the struct will also differ and the two calls will not be dependencies. However, due to the imprecision of static analysis, MoonShine always treats these calls as dependencies irrespective of their arguments.

7.2 Future Work

Supporting other Kernel Fuzzers. Most fuzzers, irrespective of their design, benefit significantly from using a diverse and compact set of seeds [31]. MoonShine's trace distillation mechanism is designed to increase the diversity and minimize the size of seed traces (while maintaining the dependencies) used for kernel fuzzing. Although our current prototype implementation is based on Linux and Syzkaller, there are a several ways we can extend MoonShine to benefit other kernel fuzzers. In particular, for other Linux kernel fuzzers, it should be relatively straightforward to adapt MoonShine's trace generation and seed selection components. MoonShine's static implicit dependency analysis can also be easily extended to other open source OS kernels such as FreeBSD.

For closed-source operating systems like Microsoft Windows, MoonShine can potentially support trace distillation of by leveraging recent works [29, 33] using virtualization-based approaches to capturing system call traces and kernel code coverage albeit with higher performance overhead. MoonShine can be extended to dynamically identify implicit dependencies by tracking the load and store instructions executed during a system call execution and identifying the calls that read-/write to the same addresses. Such a virtualization-based dynamic approach to tracking implicit dependencies will be more precise (i.e., fewer false positives) than MoonShine's static-analysis-based approach, but will incur significantly higher performance overhead. Exploring this tradeoff is an interesting area for future research.

Fuzzing Device Drivers. The system calls in our traces targeted core subsystems of the Linux kernel such as file system, memory management, and networking. However, device drivers make up over 40% of the Linux source code [15] and are the most common source of vulnerabilities [34]. Recent work [20, 28] has shown that targeted fuzzing of device drivers is effective at discovering critical security vulnerabilities. We believe that these approaches can also benefit from MoonShine's trace distillation. For example, seeds distilled from traces of Android applications/services that communicate with different device drivers can be used for efficient fuzzing of Android device drivers.

8 Related Work

Seeding and Distillation. Seed selection was first explored in the context of file-format fuzzing, i.e., fuzzers for application code that parse well-structured input (pdfs, jpeg, png, etc.). In 2008, Ormandy et al. seeded a fuzzer for the Microsoft internet explorer browser with contents gathered by crawling different URLs and uncovered two serious security vulnerabilities [27]. In 2011, Evans et al. also seeded a fuzzer for Adobe Flash Player with 20,000 distilled SWF files and discovered 400 unique crashes [19].

Recently, Beret et al. evaluated four distillation strategies on the CERT Basic Fuzzing Framework (BFF) [2] across 5 file formats and found maximizing code coverage to be the optimal distillation strategy [31]. While MoonShine is also a seed distillation framework, distillation for OS fuzzers is fundamentally a different and arguably more difficult problem than distilling file formats. File-format distillation works at the level of entire files and simply selects a small set of seed files out of a given set of files without worrying about pruning each individual file's contents. By contrast, OS fuzzer distillation must work at the finer granularity of individual system calls within program traces and maintain the implicit/-explicit dependencies of the system calls while minimizing the number of calls as the program traces tend to be, on average, multiple orders of magnitude larger than the

seed files used for fuzzing.

Seed Generation and Generational Fuzzers. Generational fuzzers craft test inputs according to some form of specification and are often used to fuzz programs which take highly-structured input, e.g., compilers. For instance, jsfunfuzz [32], and Csmith [40] are equipped with JavaScript and C grammars, respectively, which they use to craft syntactically valid programs. Other fuzzers use dynamically learned grammars to help craft input. For example, Godefroid et al. [21] present a white-box fuzzer which generates grammatical constraints during symbolic execution.

Another related line of work has investigated the possibility of synthetically crafting new seeds from existing ones. LangFuzz [24] and IFuzzer [36] are both JavaScript fuzzers that parse code fragments from an input test suite and recombine these fragments to craft interesting new inputs. Skyfire [39] uses a PCSG (probabilitistic context-sensitive grammar) learned from input programs to generate diverse and uncommon seeds. By contrast, MoonShine distills the seed traces while preserving both syntactic and semantic integrity and the achieved code coverage.

Lastly, IMF [22] is a model-based macOS kernel fuzzer that programatically infers an API model from the call trace of real-world programs. Using this inferred model, IMF is able to generate and mutate C programs for use in a fuzzing campaign. Both IMF and MoonShine rely on tracking explicit input dependencies between system calls. However, unlike MoonShine, IMF does not perform any trace distillation, which in our setting slows the rate of fuzzing by up to 90%. Furthermore, IMF does not support any implicit dependency tracking, which was essential for finding 10 out of the 17 vulnerabilities detected by MoonShine.

Other Fuzzers. Trinity [11], iknowthis [4], and sysfuzz [9] are other examples of Linux system call fuzzers built with hard-coded rules and grammars. In addition, there also exists another class of evolutionary kernel fuzzers built on or inspired by AFL [1]. These are TriforceLinuxSyscallFuzzer [10], TriforceAFL [23], and kAFL [33], the latter two of which are OS agnostic. Like Syzkaller, all of these OS fuzzers can potentially benefit from the coverage improvements offered by the MoonShine framework.

Finally, the class of evolutionary fuzzers that target semantic bugs (e.g., SlowFuzz [35], NEZHA [30], Frankencerts [17], and Mucerts [41]) may also similarly benefit from domain-specific seed distillation techniques that maximize coverage or path diversity.

Implicit Dependencies. MoonShine's approach of identifying implicit dependencies across system calls is conceptually similar to the dependency tracking mechanisms used in record-replay systems that can replay an application's execution trace. Deterministic replay requires identification of the system calls that access some shared resources to ensure preserving their relative ordering during replay. To do this, record-replay systems like Dora [25] and Scribe [26] log serialized access to shared kernel resources, e.g., inodes and memory tables. However, MoonShine, unlike these systems, uses static analysis to track implicit dependencies.

9 Developer Responses

We have responsibly disclosed all the vulnerabilities identified in this work to the appropriate subsystem maintainers and vendors. In total, 9 of the 17 vulnerabilities have already been fixed and we are working with the developers to fix the rest. Our reports include a description of the bug, our kernel configs, and a Proof-of-Concept (POC) test input. The `inotify` buffer overflow vulnerability was assigned CVE-2017-7533 and the fix was applied to the 4.12 kernel and backported to all stable kernels versions after 3.14. The JFS memory corruption and `socket_setattr` bugs were addressed within a week of disclosure and have been assigned CVE-2018-12233 and CVE-2018-12232 respectively. The fixes for both of these bugs are currently being tested and will be backported to the affected stable kernels after the 4.18-rc2 release.

10 Conclusion

In this paper we designed, implemented and evaluated Moonshine, a framework that automatically generates seeds for OS fuzzers by distilling system call traces gathered from the execution of real programs. Our experimental results demonstrated that Moonshine is able to efficiently distill a trace of over 2.8 million system calls into just over 14,000 calls while preserving 86% of the coverage. Moreover, the seeds generated by Moonshine improved the coverage of Syzkaller by over 13%, and resulted in the discovery of 17 new vulnerabilities in the Linux kernel that the default Syzkaller could not find by itself.

11 Acknowledgement

We would like to thank Dan Carpenter for his prompt and helpful responses to all our Smatch-related questions. We also thank Junfeng Yang, Jason Nieh, and the anonymous reviewers for their constructive and valuable feedback. This work is sponsored in part by NSF grant CNS-16-17670, ONR grant N00014-17-1-2010, and a Google Faculty Fellowship. Any opinions, findings, conclusions, or recommendations expressed herein are those

of the authors, and do not necessarily reflect those of the US Government, Google, ONR, or NSF.

References

[1] afl. https://github.com/mirrorer/afl.

[2] CERT Basic Fuzzing Framework (BFF). https://github.com/CERTCC-Vulnerability-Analysis/certfuzz.

[3] Glibc Testsuite. https://sourceware.org/glibc/wiki/Testing/Testsuite.

[4] iknowthis. https://github.com/rgbkrk/iknowthis.

[5] Kernel memory leak detector. https://www.kernel.org/doc/html/v4.10/dev-tools/kmemleak.html.

[6] Linux Kernel Selftests. https://www.kernel.org/doc/Documentation/kselftest.txt.

[7] Linux Testing Project. https://linux-test-project.github.io/.

[8] Open Posix Test Suite. http://posixtest.sourceforge.net/.

[9] sysfuzz: A Prototype Systemcall Fuzzer. https://events.ccc.de/congress/2005/fahrplan/attachments/683-slides_fuzzing.pdf.

[10] Triforce Linux Syscall Fuzzer. https://github.com/nccgroup/TriforceLinuxSyscallFuzzer.

[11] Trinity. https://github.com/kernelslacker/trinity.

[12] afl-tmin. http://www.tin.org/bin/man.cgi?section=1&topic=afl-tmin, 2013.

[13] Strace. https://strace.io/, 2017.

[14] The Undefined Behavior Sanitizer - UBSAN. https://www.kernel.org/doc/html/v4.11/dev-tools/ubsan.html, 2017.

[15] S. Bhartiya. How Linux is the Largest Software Project. https://www.cio.com/article/3069529/, 2016.

[16] N. Brown. Smatch: pluggable static analysis for C. https://lwn.net/Articles/691882/, 2016.

[17] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov. Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, pages 114–129, 2014.

[18] J. Edge. The kernel address sanitizer. https://lwn.net/Articles/612153/, 2017.

[19] C. Evans, M. Moore, and T. Ormandy. Fuzzing at scale. https://security.googleblog.com/2011/08/fuzzing-at-scale.html.

[20] DIFUZE: Interface Aware Fuzzing for Kernel Drivers. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 2123–2138, 2017.

[21] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 206–215, 2008.

[22] H. Han and S. K. Cha. IMF: Inferred Model-based Fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2345–2358, 2017.

[23] J. Hertz and T. Newsham. Project triforce: Run afl on everything! https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything/, 2017.

[24] C. Holler, K. Herzig, and A. Zeller. Fuzzing with Code Fragments. In *Proceedings of the 21st USENIX Security Symposium*, pages 445–458, 2012.

[25] N. Viennot, S. Nair, and J. Nieh. Transparent Mutable Replay for Multicore Debugging and Patch Validation. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 127–138, 2013.

[26] O. Laadan, N. Viennot, and J. Nieh. Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems. In *Proceedings of the ACM (SIGMETRICS) International Conference on Measurement and Modeling of Computer Systems*, pages 155–166, 2010.

[27] T. Ormandy. Making software dumber. http://taviso.decsystem.org/making_software_dumber.pdf, 2008.

[28] P. Paganini. Google syzkaller fuzzer allowed to discover several flaws in linux usb subsystem. https://securityaffairs.co/wordpress/65313/hacking/linux-usb-subsystem-flaws.html, 2017.

[29] J. Pan, G. Yan, and X. Fan. Digtool: A Virtualization-Based Framework for Detecting Kernel Vulnerabilities. In *Proceedings of the 26th USENIX Security Symposium*, pages 149–165, 2017.

[30] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana. NEZHA: Efficient Domain-Independent Differential Testing. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P)*, pages 615–632, 2017.

[31] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley. Optimizing Seed Selection for Fuzzing. In *Proceedings of the 23rd USENIX Security Symposium*, pages 861–875, 2014.

[32] J. Ruderman. Introducing jsfunfuzz. http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/, 2007.

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

[34] J. V. Stoep. Android: protecting the kernel. Linux Security Summit, 2016.

[35] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2155–2168, 2017.

[36] S. Veggalam, S. Rawat, I. Haller, and H. Bos. IFuzzer: An Evolutionary Interpreter Fuzzer using Genetic Programming. In *Proceedings of the 21st European Symposium on Research in Computer Security (ESORICS)*, pages 581–601, 2016.

[37] D. Vykov. Syzkaller. https://github.com/google/syzkaller, 2016.

[38] D. Vyukov. Kernel: add kcov code coverage. https://lwn.net/Articles/671640/, 2016.

[39] J. Wang, B. Chen, L. Wei, , and Y. Liu. Skyfire: Data-Driven Seed Generation. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P)*, pages 579–594, 2017.

[40] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Languge Design and Implementation (PLDI)*, pages 283–294, 2011.

[41] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao. Coverage-directed differential testing of JVM implementations. In *Proceedings of the 37th ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 85–99, 2016.