SyzGrapher: Resource-Centric Graph-Based Kernel Fuzzing

Marius Fleischer*,[1,2], Harrison Green†,[1], Ilya Grishchenko‡,[3], Christopher Kruegel§, Giovanni Vigna§
*NVIDIA, Santa Clara, CA, USA
†Carnegie Mellon University, Pittsburgh, PA, USA
‡University of Toronto, Toronto, Ontario, Canada
§University of California, Santa Barbara, Santa Barbara, CA, USA

***Abstract*—The operating system kernel manages system resources and makes them available to user-space programs through system calls (syscalls). Vulnerabilities in this syscall handling code could allow user-space programs to exploit the kernel, leading to information leaks or privilege escalation. Finding and patching kernel bugs is therefore critical for system security.**

Coverage-guided kernel fuzzers such as SYZKALLER have proven to be quite effective at discovering kernel bugs through mutation and generation of syscall sequences. Recent fuzzers have integrated techniques for learning dependency relations between syscalls in order to increase the efficacy of fuzzing.

Our tool, SYZGRAPHER, extends this vein of research, aiming to capture the *semantics* of syscalls to construct test cases that reach deep, interesting code. We focus specifically on improving handling of kernel *resources*, such as file descriptors and sockets. We design and implement an analysis to learn fine-grained, *resource-based* dependencies between syscalls and integrate these learned dependencies into a fork of SYZKALLER with resource-centric, graph-based mutations. Our evaluation demonstrates that SYZGRAPHER achieves more code coverage and finds more bugs than state-of-the-art tools. Additionally, in a 7-day fuzz campaign, SYZGRAPHER found 38 new vulnerabilities across 4 versions of the Linux kernel, 16 within the first day of fuzzing.

I. INTRODUCTION

The operating system kernel is the trusted part of the operating system that operates at a higher privilege level than ordinary users and is responsible for managing system resources. In order to interact with the kernel, user-space programs communicate through system calls, often shortened to *syscalls*. The system call interface acts as a crucial security boundary between unprivileged users and the privileged core runtime of the OS.

In the standard threat model for kernels, an attacker can execute unprivileged user-space code and invoke arbitrary syscalls. A sequence of system calls that triggers a bug in the syscall handling code may result in a vulnerability that allows the attacker to leak information or elevate their privilege level. Finding and patching these kernel bugs before attackers can exploit them is therefore paramount.

While there have been many proposed approaches to find (kernel) bugs [1], [2], [3], [4], [5], in recent years, *coverage-guided fuzzing* has emerged as an effective technique. Kernel fuzzers such as SYZKALLER [6] work by synthesizing test cases containing short sequences of system calls and evaluating them against the kernel, checking for crashes. Typically, feedback from the kernel, such as the coverage of edges between basic blocks of binary code, is used to guide the fuzzing process: test cases that reach new code in the kernel are marked as interesting and saved in a corpus. Later, the fuzzer can mutate these inputs to generate variants that are likely to further increase coverage.

While, in theory, any sequence of syscalls could be used to trigger a kernel bug (even ones which use the API incorrectly), in practice the vast majority of these sequences are uninteresting, reaching only shallow code paths. For example, the `read` syscall in Linux expects the first argument to be a file descriptor `fd` represented by an integer. The kernel will accept any integer for this argument, but integers that do not refer to an open file descriptor will quickly hit an error path, causing the syscall handler to exit early.

Indeed, most recent approaches to kernel fuzzing propose methods to modify the search space of the fuzzer to focus on "more interesting" syscall sequences: SYZKALLER [6] defines a grammar for system calls enabling it to synthesize the expected memory structures and pass resources between calls; MOONSHINE [3] builds a *distilled* corpus of inputs by learning syscall relations from real program traces; HEALER [2] learns influence relations between pairs of syscalls that depend on each other and uses this information during mutation; and ACTOR [1] identifies *actions* that syscalls perform in order to generate sequences that are more likely to exhibit interesting behavior.

In this paper, we improve the syscall sequence quality by constructing *semantically correct* syscall sequences whenever possible. While this may miss crashes caused by nonsensical sequences, we empirically find that test cases that follow certain semantic rules are more effective at finding new and deeper coverage.

Specifically, in our work, we improve how fuzzers handle *kernel resources* – handles such as `fd` and `sock` that are passed between syscalls to refer to kernel objects. In contrast to prior works such as SYZKALLER, which only supports random or manually-defined resource passing, we focus on automated identification of semantically correct syscall sequences manipulating resources.

Previous research in the kernel fuzzing area has successfully applied *dependency analysis* between syscalls (either static or dynamic) to improve fuzzing effectiveness [1], [2], [3], [6]. A commonality among all these works however, is

[1]These authors contributed equally to this work.
[2]This work was performed when the author was a student at University of California, Santa Barbara.
[3]This work was performed when the author was a postdoctoral researcher at University of California, Santa Barbara.

that the discovered dependencies are *global*, that is a major assumption is that one syscall changes the *global* state, e.g., via manipulation of a global state variable, in a way that is relevant for another syscall. While global dependencies are useful, we observe that many state changes also happen in a way that is local to *resources*. For example, after creating a socket (a `sock` resource), syscalls that reference this socket (such as `bind` or `connect`) can modify the state of *this socket* without affecting any other sockets that may also exist. Changing the arguments to `bind` and `connect` (i.e., which resource we pass in) will therefore affect *which socket* will get modified.

Therefore, many dependencies exist that depend not only on the *order* of the two syscalls involved (as identified by other state-of-the-art tools), but on specifically how resources are passed from the first syscall to the second. Test cases that use the wrong resources, invalid resources, or otherwise pass resources between calls incorrectly are much less likely to exercise interesting code, and much more likely to hit shallow error paths. Our work focuses on both learning and enforcing correct ways to handle resources in order to spend more time fuzzing interesting code, and less time getting stuck in these error paths.

Based on these observations, we design our resource-centric, graph-based kernel fuzzer: SYZGRAPHER. Our system first performs a one-time, static dependency analysis on the kernel code in order to learn a set of resource-based dependencies between syscalls. In order to utilize this information at runtime, we extend SYZKALLER to support graph-based mutations that sample from these dependencies while preserving the structure of test cases.

Our dependency analysis is motivated by the observation that many resource-based dependencies are implemented through state variables. For example, syscall A may *set* `state` to `ACTIVE` and syscall B may *check* if `state` is `ACTIVE`. We automate this analysis by designing and implementing a static system that identifies these pairs of *setters* and *checkers* for the same resource *values*.

While SYZKALLER supports various types of program mutations, it is not currently able to use such dependency information at runtime. Therefore, we extend SYZKALLER's mutation engine with a series of graph-based mutations (similar to GRAPHFUZZ [7]) where the resource graph is explicitly modeled. We make several significant changes to the original graph mutation algorithms in order to incorporate our dependency analysis and handle new challenges that arise in kernel fuzzing. Our modified mutation engine is able to sample from these dependencies efficiently at runtime and generate mutated test cases that express resource-based dependencies with higher probability.

In this paper, we make the following contributions:

- We designed and implemented an analysis which discovers resource-based dependencies between pairs of syscalls.
- We built SYZGRAPHER: an extension of SYZKALLER that uses *graph-based mutators* to synthesize test cases which express these resource-based dependencies with higher probability.
- We evaluated SYZGRAPHER against the state-of-the-art kernel fuzzers, where it obtained the most coverage and found the most bugs (30) in 24-hours, of which 8 were not detected by any other tool.
- We ran a bug-finding campaign on the recent kernel versions and identified 38 zero-days, 16 within the first 24 hours. We responsibly reported 9 of our findings to the responsible parties and one has already been confirmed.

SYZGRAPHER is available as open source[1].

II. BACKGROUND

A. SYZKALLER

One of the most popular kernel fuzzers (both in industry and academia) is SYZKALLER [6], a coverage-guided kernel fuzzer that uses a custom syscall specification language `Syzlang` to generate sequences of syscalls with correctly structured arguments. SYZKALLER has been deployed at scale (as SYZBOT) for more than 7 years [8] and runs continuously to find bugs in the Linux kernel (as well as other targets). Most previous work on kernel fuzzing aims to improve either parts of SYZKALLER's fuzzing loop [9], [5], [2], [3], [1] or automate manual tasks related to deploying the fuzzer [10], [11].

Fuzz Loop. At a high level, SYZKALLER works by maintaining a *corpus* of test cases (sequences of syscalls along with arguments). Programs are sampled from this corpus and *mutated* in order to generate new programs, which are then invoked against the kernel. Programs that reach new code are deemed *interesting* and added to the corpus. Also, occasionally, SYZKALLER will choose to *generate* new programs from scratch to test instead of picking one from the corpus.

Syscall Specification. A key component of SYZKALLER is the custom syscall specification language `Syzlang`, which describes the expected structure of inputs to every syscall. This specification is written and maintained by the developers of SYZKALLER and consists of a list of syscalls along with the structure of every expected argument. These arguments are composed of common data structures such as primitive types, structs, unions, arrays, and pointers.

Resources. `Syzlang` also defines *resources* in order to model values that can be passed between syscalls. For example, an open file in Linux is represented by an integer *file descriptor* or `fd`. Internally, the kernel uses this integer as a handle to retrieve the relevant context structures. In `Syzlang`, the line `resource fd[int32]` defines the resource `fd` with the type `int32`. Syscalls that take `fd` as an argument, such as *read*, *write* and *close* can therefore sometimes receive the value from syscalls that return `fd`, such as *open*.

It is also possible to specify inheritance with resources. The `fd_msr` resource, for example, represents a specific type of

[1]https://github.com/ucsb-seclab/syzgrapher

file descriptor opened on one of the /dev/cpu/#/msr files, which provide an interface to the model-specific register (msr) driver of an x86 CPU. In Syzlang, this resource is defined as a subtype of fd (resource fd_msr[fd]) so it can be used by syscalls which take *precisely* a fd_msr (such as ioctl$X86_IOC_RDMSR_REGS), but also by syscalls which take any sort of fd (such as read and write). Similarly, sock (representing a socket) inherits from fd while both sock_in (an IPv4 socket) and sock_in6 (an IPv6 socket) inherit from sock. These sorts of patterns are used extensively across Syzlang definitions. A more complete snippet of these examples is shown in Figure 1.

```
resource fd[int32]
resource fd_msr[fd]
resource sock[fd]
resource sock_in[sock]
resource sock_in6[sock]

open(file ptr[in, filename], flags flags[open_flags],
  mode flags[open_mode]) fd
read(fd fd, buf buffer[out], count len[buf])
close(fd fd)
ioctl$X86_IOC_RDMSR_REGS(fd fd_msr, cmd const[X-
  86_IOC_RDMSR_REGS], arg ptr[in, array[int32, 8]])
socket$inet(domain const[AF_INET], type flags[socket_-
  type], proto int32) sock_in
```

Fig. 1. Example snippet of Syzlang as used by SYZKALLER.

B. Resource-Based Dependencies

Resources in the Linux kernel are *stateful* objects. Certain syscalls can change the state of resources while other syscalls depend on the resource being in a particular state in order to execute properly.

In some cases, resources contain explicit *state* fields in their context structures. For example, struct socket in Linux has a field state, which can take values such as SS_CONNECTED or SS_UNCONNECTED. In other cases, there are no explicit state fields; rather, the resource state is inferred from the values of one or multiple fields. Typically, fields associated with the state of a resource are of type integer, enum(eration), Boolean, or pointer [9].

Many syscalls require resources passed to them as arguments to be in a specific state to execute successfully. For example, it may be required to initialize a device driver resource before one can invoke do_operation. In this case, the initialize syscall may transition the resource state from the INACTIVE to ACTIVE state such that do_operation can succeed. We call this type of dependency a *resource-based* dependency between initialize and do_operation. Identifying and then leveraging these dependencies (for more effective fuzzing) is the core focus of this work.

III. MOTIVATION

In the context of SYZKALLER fuzzing, inputs are sequences of up to 40 syscalls. At the lowest level, invoking a syscall is just a single CPU instruction (e.g., syscall on x86 or int 80h interrupt), and arguments are passed via registers or the stack. Without an understanding of the expected arguments, a fuzzer that simply invokes syscall with random registers and memory values would be very unlikely to hit any valid syscalls and, therefore, unlikely to reach new code coverage.

Modeling Syntax. SYZKALLER addressed the syntax specification problem by defining the syscall interfaces and expected argument structures in Syzlang (Section II-A). These definitions allow SYZKALLER to construct sequences of syscalls with correctly structured arguments and even pass resources between syscalls correctly (as described previously). However, we observed that SYZKALLER-generated programs occasionally violate these specifications. For instance, we found cases where (1) a more general resource such as fd was used in a syscall expecting a specific argument such as fd_msr, or (2) the wrong resource was passed to a call (e.g., a sock instead of a fd_msr). Test cases that contained one of these errors were less than half as likely to find new coverage, likely due to hitting shallow error paths. A mutation engine which more accurately adheres to the specification could therefore drive exploration into new and deeper code regions.

Modeling Resource-Dependencies. Even when passing the correct types of arguments to syscalls, there are still invalid ways to invoke these calls, resulting in early exits and, therefore, no new code coverage. In a trivial example, we found cases where SYZKALLER would invoke close on a file descriptor and then continue to use this file descriptor as arguments in subsequent calls. Very likely, the close call puts the file descriptor in a state where subsequent calls will exit early. We extended this result further by manually exploring several syscalls in the network stack of the Linux kernel (such as the one described in the next section). With a preliminary list of resource-based dependencies, we analyzed the programs generated by a normal SYZKALLER fuzz campaign and found that those programs that *naturally* satisfied certain dependency relations had a higher chance of generating new coverage on average. Therefore, our hypothesis is that generating these dependencies *intentionally* and *more frequently* would be very beneficial for fuzzing.

Dependency Example. As an example of a resource-based syscall dependency found in the Linux kernel, consider the code snippets from accept and listen shown in Figure 2. During the execution of the syscall listen, the kernel reaches the function inet_csk_listen_start (Line 1). Among other tasks, this function writes the value TCP_LISTEN to the field sk_state of the socket by calling inet_sk_state_store (Line 5). Later on, when the user calls the syscall accept, the kernel checks in inet_csk_accept (Line 9) if the socket is in the state TCP_LISTEN before proceeding (Line 15). The OS proceeds with the execution of the syscall only if the correct value is found and the dependency is satisfied; otherwise, the OS returns an error. Only by fulfilling the dependency of calling listen before accept, we can avoid this shallow error path.

Avoiding these invalid sequences (and therefore synthesizing *valid* ones) requires a stronger understanding of the intended *semantics* of the syscalls – i.e., the intended ways to call them. In this work, we focus on capturing such semantics by discovering dependencies between syscalls. We define each such dependency as follows.

Resource-Based Dependency. A syscall B depends on a syscall A if (1) A and B each have a resource argument of the same type, resource R_A in A and resource R_B in B; (2) A sets field f of a resource's struct R_A to value v; (3) B makes a control-flow decision based on whether f holds the value v in resource R_B; (4) A sets the value v along a normal execution path (i.e., non-error path) and one of the paths after the control-flow decision in B leads to the error. We call A the *setter*, as it sets the field f to the value v, and B the *checker*, as it (later) validates or checks the value of f. We refer to v as *dependency value*.

Previous work defined dependencies in multiple ways: Dependent syscalls (1) access the same state variables identified via static analysis (StateFuzz [9]), access the same memory location (2) determined by likely invariants (InvsCov [12]), or (3) identified via locality-sensitive hashing (StateAFL [13]). Instead of variables and memory locations, this work's definition focuses on concrete values assigned as resource states, specifically restricting dependencies by requiring the absence of error paths for the setter and their presence in the checker. This decision is motivated by the fact that previous work is generally interested in the impact of statefulness on any control flow (e.g., obtaining new error paths), whereas our work specifically targets dependencies that may result in shallow paths if ignored. Additionally, it allows our analysis to scale more easily to the whole kernel as it needs to analyze a smaller set of pairs of write operations and control-flow decisions. The main aspect contributing to the scalability in our analysis is that we only consider concrete value dependencies established through assignments and conditionals between syscalls manipulating resources of the same type. This relieves us of the need to perform expensive analyses, e.g., symbolic execution [9], and limits the number of syscall pairs to analyze.

IV. Approach

In this section, we describe the design details of SyzGrapher, starting with the design goals and challenges, followed by detailed descriptions of all its components.

The main design goal of SyzGrapher is to enable more effective fuzzing by preventing the fuzzer from wasting computational resources through invalid uses of resources and syscalls. Such uses typically lead to syscalls producing an error immediately. This prevents the fuzzer from meaningfully exploring the kernel's code base. Unfortunately, current state-of-the-art kernel fuzzers are mostly unaware of resource-based dependencies. Moreover, their sequence-based input-generation algorithms do not allow for the seamless incorporation of such information, even if it were available.

We design a scalable static analysis to automatically extract resource-based dependencies between syscalls. Additionally,

```
int inet_csk_listen_start(struct sock *sk)
{
   // ...
   // writing the dependency
   inet_sk_state_store(sk, TCP_LISTEN);
   // ...
}

struct sock *inet_csk_accept(struct sock *sk,
        int flags, int *err, bool kern)
{
    // ...
    error = -EINVAL;
    // dependency check
    if (sk->sk_state != TCP_LISTEN)
        goto out_err;
    // normal execution
}
```

Fig. 2. Real-world syscall dependency between `listen` and `accept` from the Linux kernel.

to integrate this information effectively into the fuzzing process, we adapt the graph-based fuzzing approach of GraphFuzz [7] to the domain of kernel fuzzing. The adapted graph-based input generation efficiently constructs highly structured inputs that respect the resource-based dependencies between syscalls. As a result, SyzGrapher is able to find new bugs in deep kernel code rather than attempting to "unlock" such code by randomly finding valid syscall sequences and proper resource usage.

A. Overview

Figure 3 shows the high-level workflow of SyzGrapher. The Steps ❶–❸ extract the dependency information from the target kernel and provide it to the fuzzer. The graph-based fuzzing technique in Step ❹ integrates the dependency information from the previous steps into the fuzzing process.

Initially, SyzGrapher is given the `Syzlang` syscall specification and uses it to extract a list of interesting *resources* and their corresponding constructors. Currently, we only consider file descriptor and socket resource types. Note, however, that this includes all resource types that are a subtype of `fd` and `sock` (more than 65% of the total number of resources). These are the most widely used types and constitute the arguments for more than 90% of syscalls in `Syzlang`.

In the first step, SyzGrapher needs to identify all parts of the kernel code that are responsible for operating on and manipulating interesting resources ❶. This is necessary so we can identify all the code that might introduce dependencies between syscalls. As discussed in Appendix A, a major challenge is the identification of indirect call targets in the kernel code. To resolve the targets of indirect calls, SyzGrapher invokes a constructor for each resource type. The resource's constructor assigns the (concrete) values of function pointers, which are used later during the dispatch. Our system reads these values from the memory and maps them to the resource-handling code using debug information.

Given a view (that is as complete as possible) of the kernel code that handles resources, SyzGrapher checks all pairs of syscalls that operate on the same resource type. The goal is to

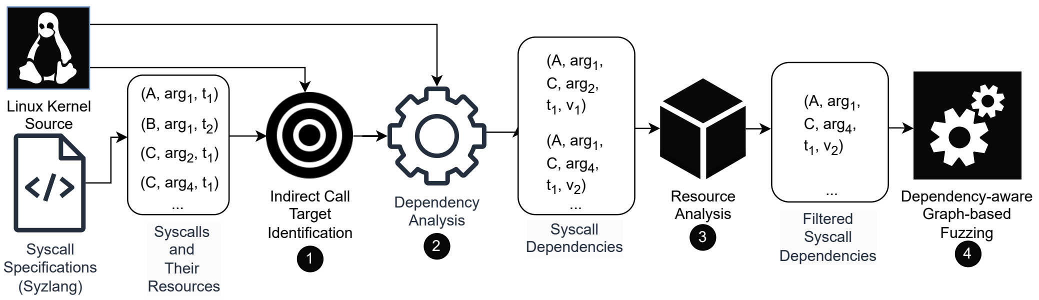

Linux Kernel Source
Syscall Specifications (Syzlang)
(A, arg1, t1)
(B, arg1, t2)
(C, arg2, t1)
(C, arg4, t1)
...
Syscalls and Their Resources
Indirect Call Target Identification
1
Dependency Analysis
2
(A, arg1, C, arg2, t1, v1)
(A, arg1, C, arg4, t1, v2)
...
Syscall Dependencies
Resource Analysis
3
(A, arg1, C, arg4, t1, v2)
...
Filtered Syscall Dependencies
Dependency-aware Graph-based Fuzzing
4

Fig. 3. The workflow of SyzGrapher.

analyze their implementation to find resource-based dependencies ❷. To this end, we developed a static analysis over the abstract syntax tree (AST) of the kernel code that identifies syscall dependencies based on concrete value dependencies (as discussed in Section III). Static analyses typically struggle to scale to code bases as large as OS kernels [14]. To avoid scalability issues, we design our analysis to first extract pairs of syscalls manipulating the same resource type. This allows us to analyze each pair independently.

The next step of the dependency analysis refines dependencies related to syscalls that involve *multiple* resources, either as arguments (for example, syscalls that receive multiple file descriptors as input arguments) or as return values ❸. In such cases, the initial dependency analysis cannot determine which *specific* arguments are responsible for introducing a dependency. To resolve this ambiguity, we perform a global taint analysis from the syscall arguments to the code location(s) that create(s) the dependency.

Finally, we provide the graph-based fuzzer with the dependency information ❹. Our graph-based approach enables SyzGrapher to effectively integrate the dependency information into the fuzzing process. The fuzzer assigns weights to each dependency. The higher the weight, the more likely the dependency will be used by the fuzzer. During the weights' assignment, rare dependencies (those with rare dependency value v, see definition in Section III) are prioritized. This follows the insight that rare dependencies are more challenging to find and test than common ones.

B. Dependency Analysis

Our dependency analysis starts by extracting information about *resources* and *syscalls* from the provided `Syzlang` specification. For each syscall, we keep track of *which* arguments are resources (and their types), and whether the call also *returns* a resource. Additionally, for every resource type, we also find a constructor syscall for this resource from the specification (i.e., a syscall that produces this resource). These constructors will be important in a later step in order to *instantiate* corresponding resources and perform indirect call target identification.

Figure 3 gives an example of this initial extraction for three syscalls and their resources: $A(t_1)$, $B(t_2)$, and $C(..., t_1, ..., t_1)$. Syscall A has one argument that references a resource of type t_1, syscall B has one argument of type t_2, and syscall C has two arguments that reference resources, both of type t_1. For this example, we would record one tuple each for syscall A and B (containing the syscall, argument index, and resource type) and two tuples for syscall C (one for each time a resource appears in the argument list).

After extracting the syscalls and the resources they manipulate, we need to identify what parts of the kernel code are responsible for *implementing* these syscalls ❶. We detail this procedure in Appendix A.

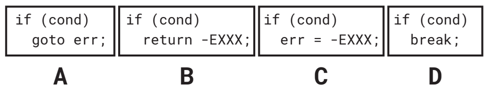
Given the code responsible for handling each resource type in the syscalls, SyzGrapher statically finds pairs of setters and checkers between pairs of syscalls ❷. CodeQL serves as the basis for this dependency analysis, combined with Python glue code for generating queries and collecting results. Our dependency definition in Section III introduces four requirements that our analysis needs to take into account.

(1) Same resource type. SyzGrapher starts by identifying all pairs of syscalls that have the same resource type in their argument list. These pairs are potential candidate *setters* and *checkers*. For the example in Figure 3, both syscall A and C use a resource of type t_1, and thus, these syscalls are selected as potential candidates.

(2) Setter. Next, SyzGrapher constructs an abstract syntax tree (AST) of the setter's syscall code and performs static analysis to identify each *assignment* operation in the AST. The aim is to find every assignment operation that assigns a *concrete* value (on the right-hand side) to any field of a struct (left-hand side).

For each such identified assignment, we record the name of the field f as well as the assigned value v. In addition to checking for direct assignments (`res.f = v;`), we also consider a common programming pattern where the kernel uses a function to perform the assignment of a value to a field (`set_f(res, v);`). Such functions typically receive only two arguments–one argument is the concrete value to assign, and the other one is (a pointer to) a struct. We find such calls in the AST and extract the value v from them.

(3) Checker. Then, SyzGrapher constructs an AST for the checker syscall and looks for *comparison operations* in the AST that involve a concrete value v and struct field f, ob-



```
if (cond)        if (cond)          if (cond)          if (cond)
  goto err;        return -EXXX;      err = -EXXX;       break;
```

Fig. 4. Common error path patterns for state validations.

served previously in the checker syscall candidate. Whenever SyzGrapher finds a combination of a setter candidate and a checker candidate (that is, a syscall that sets the value of a field f to a value v, and another syscall that includes a check for that value), it moves on to verify the last requirement.

(4) Control Flow. Finally, SyzGrapher verifies that this observed *setter-checker* pair is potentially useful for unlocking deeper coverage. Specifically, we are only interested in this pair if the setter performs the assignment in a *non-error* path (i.e., it is not performing something like `socket.state = ERR;`) and the checker will use this comparison to avoid an error path (i.e., running the setter is a *pre-condition* to a successful execution of the checker).

Identifying *error paths* in syscalls is not trivial. Most previous approaches in this area rely on computationally expensive, intra-procedural path-sensitive analyses [15], [16], [17]. To maintain scalability of our analysis, we design a lightweight analysis based on common programming patterns in the Linux kernel. We observed that, in the context of verifying resource states, the kernel uses a small set of patterns to "error out" when an invalid state is encountered. These patterns are displayed in Figure 4. Specifically, we look for places with: (A) an explicit `goto`, (B) returning a negative `E` constant, (C) setting an `err` variable, or (D) an explicit `break`. Identifying such "error out" patterns can be time-consuming, especially when aiming for completeness. However, this effort only needs to be undertaken once. Moreover, in our experiments, the current set of patterns was sufficient to demonstrate improvements over competing tools.

Based on this analysis, SyzGrapher is able to determine (approximately) whether the setter assignment is executing in a non-error path and whether the checker condition precludes an error path. If both conditions are satisfied, the dependency is saved. In our example in Figure 3, assuming these error path conditions were met and we observed A as a setter and C as a checker, SyzGrapher would produce two dependencies (since t_1 appears twice in C). Specifically, we would obtain: $(A, arg_1, C, arg_2, t_1, v_1)$ with dependency value v_1 and $(A, arg_1, C, arg_4, t_1, v_2)$ with dependency value v_2.

C. Resource Analysis

The final step of the dependency analysis phase ❸ receives all dependency tuples from the previous step and further analyzes resources which take *multiple resources* of the same type, such as `accept` or `dup2` in the Linux kernel (or syscall C in our example). For instance, if a setter syscall takes three resources of the same type but only sets a value on the *first* resource, it is important to learn this nuance such that we can properly utilize this information while fuzzing.

Therefore, SyzGrapher performs inter-procedural, field-sensitive taint tracking from each syscall argument to the specific code location – the *assignment* in the setter or the *condition* in the checker – to determine which resource is relevant. If the taint label (associated with a specific argument) reaches the code location of interest, SyzGrapher determines that this argument holds the relevant resource for the dependency. In our example (Figure 3), we assume that only resource arg_4 in C was the one with the taint reaching the check. Consequently, we see only $(A, arg_1, C, arg_4, t_1, v_2)$ tuple as the result of the dependency analysis.

Finally, SyzGrapher gives the dependency information to the graph-based fuzzing component. The output is organized as a collection of tuples D, where each tuple, as the one discussed above, is defined as (A, arg_i, B, arg_j, t, v). Intuitively, each tuple means that the call to A with a resource of type t as the value of argument arg_i must happen *before* the call to B with the resource of the same type t as the value of argument arg_j, since A sets a value v and B checks for it.

D. Graph-Based Kernel Fuzzing

While prior works [6], [2], [3], [9] focus on dependencies based on global variable or memory state (where only the *order* of two consecutive syscalls matter), we learn fine-grained *resource-based* dependencies, as described in previous sections. Syzkaller cannot naturally handle these types of dependencies: It is designed to accommodate purely ordering-based dependencies, such as "a `read` should come directly after an `open`" (and it does so through its `prio` table), but it does not attempt to handle dependencies that rely on the specific resources passed between syscalls. Importantly, with *resource-based* dependencies, the syscalls involved may not necessarily come directly after each other. For instance, a `read` may come several syscalls after an `open` on the same resource as long as the interleaving syscalls do not operate on this shared resource.

In order to visualize these types of connections between syscalls that operate on the same resource, it is natural to represent test cases not as a linear sequence but as a *directed graph* where syscalls are nodes and shared resources and edges. Figure 5 shows the same test case in the linear form used by Syzkaller (left) and the directed graph form (right). A dependency in this representation is a *setter* node which has an edge (resource) connected to a *checker* node.

Given a set of resource-based dependencies ❸, we want mutation algorithms that can *more frequently* generate directed graphs that contain these sorts of connections. Rather than reinvent these mutators from scratch, we adopt the algorithms described in GraphFuzz [7]–a structure-aware fuzzer designed to fuzz C/C++ library APIs. While GraphFuzz was designed for a different domain, the test cases were similarly directed graphs, with library functions as nodes and objects passed between functions as edges.

In our adaptation, vertices are *syscalls* and edges represent *resources* which are passed between these syscalls. Edges originate from syscalls that create resources and connect to

```
r0 = openat$nci(0xffffffffffffff9c, &(0x7f0000000080),
  0x2, 0x0)
ioctl$IOCTL_GET_NCIDEV_IDX(r0, 0x0,
  &(0x7f00000000c0)=<r1=>0x0)
r2 = syz_init_net_socket$nl_generic(0x10, 0x3, 0x10)
r3 = syz_genetlink_get_family_id$nfc(&(0x7f0000000100),
  r2)
sendmsg$NFC_CMD_DEV_UP(r2, &(0x7f0000000140)={0x0, 0x0,
  &(0x7f0000000180)={&(0x7f00000001c0)={0x1c, r3, 0x1,
  0x123, 0x234, {}, [@NFC_ATTR_DEVICE_INDEX={0x8, 0x1,
  r1}]}, 0x1c}}, 0x0)
write$nci(r0, &(0x7f0000000340)=@NCI_OP_CORE_INIT_RSP,
  0x14)
write$nci(r0, &(0x7f0000000400)=@NCI_OP_RF_DISCOVER_MAP_R-
  SP, 0x4)
```

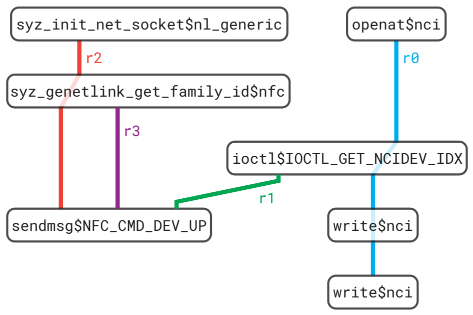

syz_init_net_socket$nl_generic
openat$nci
r2
r0
syz_genetlink_get_family_id$nfc
r3
ioctl$IOCTL_GET_NCIDEV_IDX
r1
sendmsg$NFC_CMD_DEV_UP
write$nci
write$nci

Fig. 5. A program test case viewed in textual format (left) and resource-centric graph format (right). Resources in the graph are colored in the textual format and form edges of the graph in the graph format.

syscalls that use these resources. For example, an edge from syscall A to syscall B indicates that there is some resource produced by syscall A which is then used by syscall B. If syscall B is not marked as a destructor (see Section IV-D4), this resource is also an *output* of syscall B, and can be linked to some syscall C which uses the resource next.

We utilize Syzlang definitions as a reference for the expected inputs and outputs of each syscall. While the basic concept and mutation algorithms are similar to the approach described in GRAPHFUZZ, there are some significant differences when applying the technique to kernel fuzzing that present new challenges and opportunities. We discuss most of them in the following paragraphs. The differences in requirements and threat model are detailed in Appendix B.

1) Mutation: We fork SYZKALLER (commit bf285f0) and extend its default mutation engine with a new set of graph-based mutations that allow us to incorporate our resource-based dependencies. These mutations consist of first a *mutation* that changes the structure of the graph (perhaps leaving it in an incomplete state), followed by *graph completion* (Section IV-D3), which fills in any missing inputs by introducing new nodes.

Our set of newly-added graph-based mutations consists of the following (we describe dependency-based adaptations to these in Section IV-D6):

- **Crosslink**: links the output of some syscall A to the input of some syscall B within the same program.
- **Crossover**: merge two programs from the corpus and invoke **Crosslink** with syscall A sampled from the first program and syscall B from the second.
- **Order**: without changing the connectivity, adjust tie-breaking such that syscalls may execute in a different order.
- **ReplaceConstructor**: pick an input to some syscall A and remove the subgraph that produces this resource (it will be regenerated differently during graph completion).
- **SpliceIn**: pick two syscalls A and B such that there is a resource produced by A and consumed by B; insert a new syscall C in between these syscalls and create a new link from A to C and C to B.
- **InsertNode**: insert a random new node into the program without otherwise changing the connectivity.
- **InPlace**: pick a syscall A and regenerate the non-resource arguments (such as constants and buffers); this mutator does not change the connectivity.

2) Generation: We also define a new graph-based generation routine. Specifically, our generation algorithm consists of initializing a new program with a single syscall and invoking graph completion (Section IV-D3) to create any required resource constructor. A small percentage of the time, we also invoke graph mutation at this stage to introduce more variety.

3) Graph Completion: Graph completion is the process of taking a partially complete graph (i.e., one with missing edges) and creating the requisite constructors or destructors to satisfy those missing edges. Our implementation works similarly to the one described in GRAPHFUZZ, in that we maintain a compact data structure which contains all possible ways to construct every resource. During graph completion, we then iterate over every missing input in a partial graph and sample a potential constructor from this data structure.

While many resources can be constructed with a single syscall, for example open for fd, some resources require several consecutive calls. For instance, kvm_run_ptr represents a pointer inside a virtual cpu in the KVM module and is constructed with mmap$KVM_VCPU. But in order to invoke that syscall, one needs a fd_kvmcpu which can only be constructed from a fd_kvmvm, which in turn needs a fd_kvm. During graph completion, this three-level constructor sequence can be added in a *single step*.

4) Resource Lifetime: Kernel resources are *created* by certain syscalls and *destroyed* by others. A file descriptor fd resource can be created by open and is later destroyed when passed to close. Attempting to use the resource before it has been created (i.e., referencing an unknown file descriptor) or after it has been destroyed (i.e., a closed file descriptor) is unlikely to do anything interesting. We find that SYZKALLER (sometimes intentionally) breaks both of these rules: using so-called *special values* in place of constructor syscalls and continuing to use resources even after they have been passed to destructors such as close. This is due to the fact that the syscall specifications lack information about resource destruction. Since only a small number of syscalls need to be marked as destructors, we added these annotations manually.

The graph-mutation algorithms ensure that resources that are destructed are never reused and that all resources which are inputs to syscalls are properly constructed.

5) Resource Hierarchy: Resources in the Linux Kernel exist in a hierarchy that describes inheritance relationships. For example, while `fd` represents a general file descriptor, `fd_msr` represents a *more specific* version of `fd` that has been obtained by invoking the syscal `open` on a `/dev/cpu/#/msr` file. In SYZKALLER's `Syzlang`, certain syscalls are defined to work on any `fd` while others are defined *specifically* for `fd_msr`. In other words, a syscall that takes an `fd` can also work with any child resource (like `fd_msr`). However, syscalls defined on `fd_msr` can not take a general `fd`.

In order to preserve type validity while mutating programs, we perform a lightweight analysis, looking at *all* usages of a given resource in order to determine the *compatible* syscalls according to the resource hierarchy. An example of this process is shown in Figure 6. In this example, `A` is the most general resource, `B` and `E` inherit from `A`, `C` and `F` inherit from `B` and `E` respectively, and `D` inherits from `C`. The resource hierarchy tree is displayed in the center.

On the left side, we are trying to extend the graph in the forward direction (appending a syscall). The upstream constructor (`new_C`) enforces that any subsequent syscalls must take resource `C` or anything more general. In this case, one of `use_A`, `use_B`, or `use_C`.

On the right side, we are trying to extend the graph in the reverse direction (prepending a syscall). This type of extension occurs frequently during graph completion (Section IV-D3). Here, we need to ensure that the syscall we add introduces (constructs) a resource that will be compatible with all downstream calls. By looking only at the next call (`use_A`), we might naively use the constructor `new_A` to introduce a resource of type `A`. However, this resource would be incompatible with the downstream call `use_E`, which expects a resource *at least* as specific as `E`. Therefore, to preserve compatibility, we must use either `new_E` or `new_F`.

6) Integration of Resource-Based Dependencies: In this section, we explain in detail how the fuzzer uses the identified dependencies. SYZGRAPHER transforms the dependency pairs into a resource-centric choice-table called the graph choice-table (GCT). Given a specific syscall and resource argument, this choice-table provides a weight value for choosing the next (previous) syscall and its resource argument based on dependency information. SYZGRAPHER uses these weights for the graph-based mutation operators **SpliceIn** and **Crosslink/Crossover**, as well as for graph completion (Section IV-D3).

To compute weights in the GCT, we first compute the frequencies n_{v_i} of each observed dependency value (the value that the setter syscall assigns and the checker has in the conditional), that is, how many tuples (A, arg_i, B, arg_j, t, v_i) among all dependencies in D have value v_i. In the following

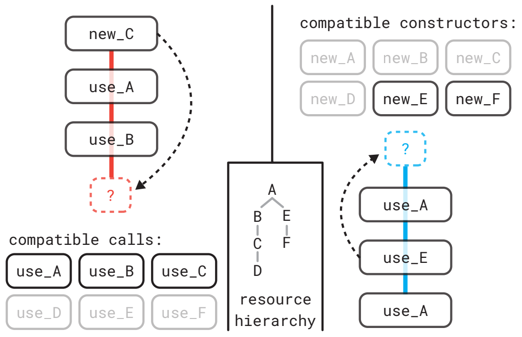


Fig. 6. Example graph extension in the forward direction (left) and reverse direction (right). The resource hierarchy tree is shown in the center. Dashed arrows show the existing syscall which enforces the constraint.

discussion, we use a different tuple representation for brevity, namely (sa_A, sa_B, v_i) with sa_A as a pair of syscall A and argument arg_i, sa_B as a pair of syscall B and argument arg_j.

Then, we intitallize weights w_v for each dependency value across all syscalls with the following formula based on inverse document frequency (IDF) [18].

$$w_v(v_i) = \frac{\frac{1}{n_{v_i}}}{\sum_j \frac{1}{n_{v_j}}} \tag{1}$$

Intuitively, the more rare the value v_i, the higher the value $w_v(v_i)$ is, because we are interested in *rare* dependencies found in only a few cases: It is hard to discover such rare dependencies by luck, thus, our dependency-aware fuzzer has a high chance of finding new coverage and bugs. Additionally, SYZGRAPHER employs a threshold t_{dep} to filter out the most frequent dependencies in D (Section IV-C) that stem from a dependency value v_i with a computed weight below t_{dep}.

For all dependencies in D, SYZGRAPHER picks a pair sa_A of syscall A and its argument arg_i, and a pair sa_B of syscall B and its argument arg_j, and calculates the weight w_{cr} by summing up the weights of all values v_i in D that apply to this pair, i.e., we consider all tuples like (sa_A, sa_B, v_i).

$$w_{cr}(sa_A, sa_B) = \sum_i w_v(v_i) \tag{2}$$

The value $w_{cr}(sa_A, sa_B)$ is a cumulative weight of all values that A sets and B checks. Finally, we construct the GCT with final weights w_f by normalizing over all setters for a particular checker.

$$w_f(sa_A, sa_B) = \frac{w_{cr}(sa_A, sa_B)}{\sum_C w_{cr}(sa_C, sa_B)} \tag{3}$$

The weight w_f is normalized by dividing the cumulative weight of all values that A sets and B checks, by the weights of all values that any other syscall C sets and B checks.

This dependency information is incorporated in several places in the graph-based fuzzing algorithms. First, to perform a **SpliceIn** mutation, SYZGRAPHER needs to select a syscall

B to insert between two other syscalls A and C. Syscall B is selected based on the GCT. That is, Bs with higher values of $w_f(sa_A, sa_B)$ and $w_f(sa_B, sa_C)$ are preferred. Second, as part of the mutation operator **Crosslink** (and by extension **Crossover**), SYZGRAPHER needs to identify a pair of syscalls in the graph that can be connected. Out of all the sets of pairs in the graph, SYZGRAPHER prioritizes pairs with higher w_f values according to the GCT. Third, when SYZGRAPHER adds syscalls to provide required resources during the graph completion step, it uses the values in the GCT for the selection of these syscalls, again prioritizing those with higher w_f weights. The integration of the GCT into the graph completion step ensures that the GCT is used during graph-based mutation even if SYZGRAPHER did not choose to perform a **SpliceIn** or **Crosslink** mutation.

V. EVALUATION

We evaluate SYZGRAPHER with the following research questions in mind. RQ1: How does SYZGRAPHER compare to the state-of-the-art kernel fuzzers? RQ2: Can SYZGRAPHER reach new parts of the codebase compared to state-of-the-art tools? RQ3: How accurate is SYZGRAPHER's static analysis? RQ4: How effective is SYZGRAPHER at finding previously unknown bugs?

Hardware Configuration. The experiments for RQ1 and RQ2 were conducted on servers with two Intel(R) Xeon(R) E5-2690 v2 @ 3.00GHz and 256 GB of RAM. The longer-running experiments for RQ4 and dependency analysis (R3) were performed on a server with two Intel(R) Xeon(R) Gold 6430 @ 3.40GHz and 1 TB of RAM. The dependency analysis was performed over a period of five days prior to the fuzzer experiments.

Fuzzer Comparison. The 24-hour fuzzer experiments (RQ1 and RQ2) were evaluated on Linux kernel version 6.8.3. For a fair comparison, we disable kernel options that need to be disabled for one of the baseline tools (ACTOR). Each fuzzing instance used 10 VMs with two cores and 8 GB of RAM. We run each tool in isolation for 24 hours and repeat each experiment five times to mitigate the effect of randomness.

Fuzz Campaign. The fuzz campaign (RQ4) was conducted on four different versions of the Linux kernel: the latest version (6.9), a stable release (6.8.3), and two long-term support releases (6.6.31, 5.15.159). At the time of our experiments, these are the versions that the Linux developers are actively maintaining and accepting bug reports for. Each fuzzing instance uses 15 VMs with two cores and 8 GB of RAM. To compile the kernel, we use the same kernel configs as SYZBOT [19] for the respective kernel versions.

A. Comparison to the State-of-the-Art (RQ1)

With RQ1, we aim to understand SYZGRAPHER's ability to find coverage and bugs in comparison to the state-of-the-art tools. For the experiment, we choose SYZKALLER, MOONSHINE, HEALER, and ACTOR as tools, for comparison for the following reasons. SYZKALLER serves as our baseline, since our tool is built on top of it. Similarly to SYZGRAPHER, MOONSHINE, HEALER, and ACTOR learn relations between syscalls. Additionally, all tools, including SYZGRAPHER, are general purpose fuzzers, rather than specialized tools for a specific part of the kernel.

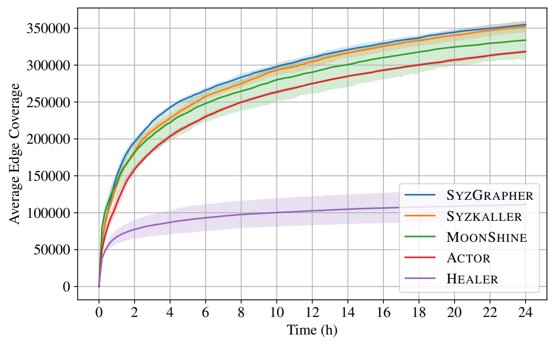


Fig. 7. Coverage trend of SYZGRAPHER, SYZKALLER, MOONSHINE, HEALER, and ACTOR over 24 hours.

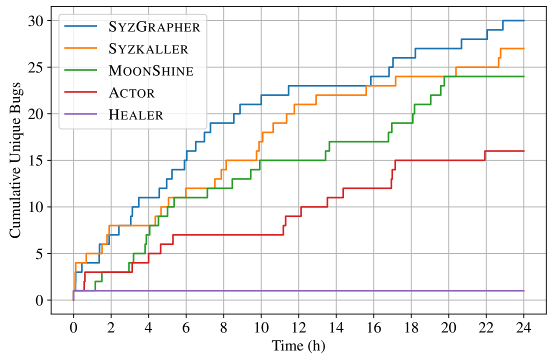


Fig. 8. Cumulative unique bugs found by SYZGRAPHER, SYZKALLER, MOONSHINE, HEALER, and ACTOR over 24 hours.

MOONSHINE [3] analyzes traces of real world programs to create a *distilled* corpus that captures dependencies found in the original traces. Unfortunately, the open-source implementation of MOONSHINE is not compatible with the latest versions of the Linux kernel [20]. Therefore, we perform the following best-effort comparison. We fallback to cross-pollination [21], a technique commonly used in fuzzing that describes the reuse of a fuzzing corpus from one target to another target with the same input format. In our case, the different targets are different versions of the Linux kernel. As the syscall interface of the Linux kernel is relatively stable across versions, we expect the corpus used by MOONSHINE in their original evaluation to largely apply to the kernel used in our experiments. For ACTOR and HEALER, we use the open-source implementations available on GitHub (note, however, that the public version of HEALER is different than the private one used for their original experiments [22]).

In Figure 7, we report the average edge coverage of each tool over the 24-hour fuzz campaign. Figure 8 shows the cumulative unique bugs found by each tool (across all five of the runs). We evaluate uniqueness by looking at the crash message produced, with specific numbers and addresses removed (i.e., the same way SYZKALLER deduplicates bugs in its reports). SYZGRAPHER obtains the most coverage on average after 24 hours, marginally beating SYZKALLER (-0.8%), and well above MOONSHINE (-5.9%), ACTOR (-10.3%), and HEALER (-68.7%). SYZGRAPHER also finds the most unique bugs (30) within 24 hours, more than SYZKALLER (27), MOONSHINE (24), ACTOR (16), and HEALER (1). Of these 30 bugs, 8 were not found by any of the other tools during the entire 24 hours.

B. Unique Coverage (RQ2)

While RQ1 demonstrated that SYZGRAPHER can find *more* bugs and edge coverage over a 24-hour period compared to other tools, we also assess its ability to find *new* and *unique* coverage. To this end, we take the final corpora generated by each tool after the 24-hour fuzz experiment (RQ1) and compute the *block* coverage for each program in the corpus.

SYZGRAPHER was able to generate testcases that reached coverage in 343 new functions across the kernel codebase. Since we have not augmented SYZGRAPHER with definitions for any *new* syscalls, this new coverage is the result of using *existing* syscalls in new ways. These new functions were found predominantly in kernel subsystems that make heavy use of resources: net (130), drivers (75), and fs (62), among others. While these subsystems are also the largest components of the Linux kernel, this result matches the intuition that SYZGRAPHER should be especially suited for finding new coverage with syscalls that rely on proper resource handling.

In order to understand this new coverage better, we explore three example programs that reach new functions. These programs are displayed in the graph representation in Figure 9 to highlight the resource connectivity, and we include the full text-based programs in Appendix C.

A: KVM. Program A (Figure 9, left) shows a test case that reached new coverage in the KVM Hyper-V component in the kernel. In this case, SYZGRAPHER was able to find a test case that properly constructed a KVM's virtual CPU and enabled the Hyper-V synthetic interrupt controller (SYNIC), before finally running the virtual CPU. In this program, the resource connectivity is critical; invoking these `ioctl` operation sequences on an *improperly typed* resource (such as a normal `fd`) or on *different resources* would not result in semantically correct behavior.

B: dma-buf. Program B (Figure 9, center) shows a test case that is able to reach new coverage in the `dma-buf` component in the kernel. In particular, this test case is able to interact with the `sw_sync` driver and construct two sync fences that are then merged with the `ioctl$SYNC_IOC_MERGE` call. While the `ioctl$SYNC_IOC_MERGE` call was not identified by our dependency analysis, *all* of the other resource links in this program were identified as having potential resource-based dependencies and thus the probability of sampling these patterns was boosted during fuzzing. Therefore, even in cases where the dependency analysis may miss some true positives, SYZGRAPHER can utilize the dependencies it did find (in this case dependencies between all the other calls) to reach new coverage.

C: net. Program C (Figure 9, right) shows a test case that reaches new coverage in the network stack. This program consists of two disconnected subprograms (based on our resource-centric view). One subprogram creates a new `SMC` socket, binds it to an address, then connects to it—later sending data with `sendto$inet`. The other subprogram constructs two other types of sockets: a `netlink` socket (with `socket$nl_route`) and an IPv6 UDP socket (with `socket$inet6_udp`). In this test case, *all* of the links between resources were identified as resource-based dependency candidates by our analysis, likely contributing to SYZGRAPHER's ability to find this new coverage during fuzzing.

C. Accuracy of the Dependency Analysis (RQ3)

For RQ3, we perform case studies on two targets: the TCP/IPv4 socket module and the UINPUT driver. For each target, we manually categorize the discovered dependencies into true positives (TP) and false positives (FP). Additionally, we analyze the target's source code to identify potentially-missed dependencies (false negatives - FN). To do so, we first manually examine all potential candidate syscalls for checks in the target's code and group these by the struct field of the resource involved in the condition. To keep the manual analysis effort feasible, we then sample a subset of 10 resource struct fields and look for corresponding setters.

Case Study 1: TCP/IPv4. The first case study focuses on the TCP/IPv4 socket module spanning 485 syscalls. The dependency analysis described in Section IV-B identifies a total of 297,610 dependencies. Of these, 949 dependencies remain after the filtering (using threshold t_{dep}) during the construction of the GCT (Section IV-D6).

Our false positive analysis of the dependencies reveals 85 true positives and 864 false positives. Our manual analysis reveals that the false positives stem from syscalls being identified as setters as they perform a function call that matches the heuristic described in Section IV-B. While the number of false positive appears large, we observe that they all have low weights. The top half of Figure 10 shows the distribution of true and false positives, bucketed into different weight ranges (these are the weights assigned to each dependency during the creation of the GCT). It can be seen that all true positives (the green bar) have high weights—likely to be considered by the fuzzer—and are perfectly separated from the false positives (the red bar) with low weights—likely to be ignored.

As part of our manual false negative analysis, we first identify checker candidates that perform conditional (comparison) operations that involve 50 different resource struct fields. Again, we sample 10 fields randomly for setter identification. The identification of setter syscalls over this restricted set reveals 613 dependencies that were missed by our static analysis. The main reason for false negatives is that the static analysis is currently only implemented for the $\neq$ comparison operator. It lacks support for other operators as well as conditionals involving bit operations. Missed dependencies, however, do

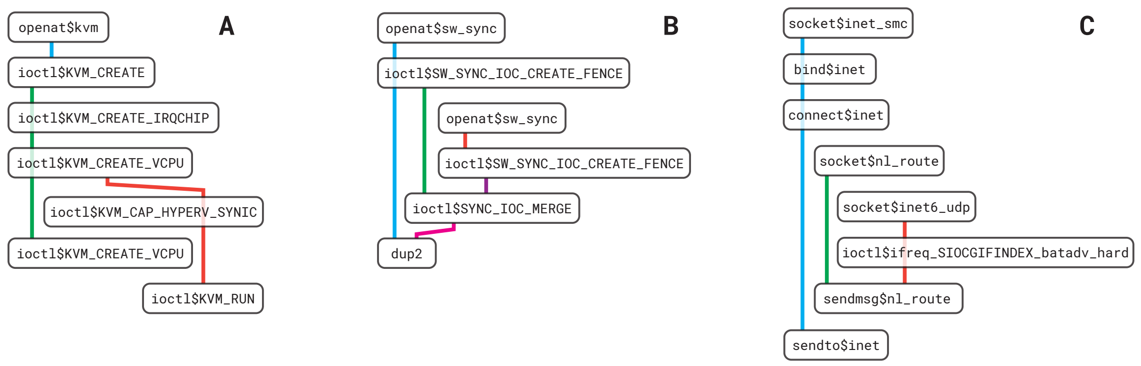


Fig. 9. Three programs from the corpus of the 24-hour SYZGRAPHER run which find unique coverage.

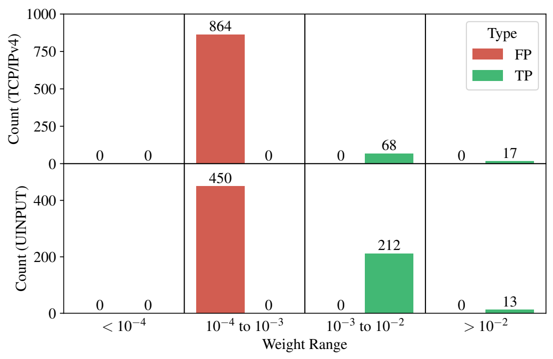


Fig. 10. Distribution of true positive (TP) and false positive (FP) dependencies for the TCP/IPv4 socket (top) and UINPUT driver (bottom).

not preclude the discovery of new coverage in meaningful ways (as we show with program B in Section V-B). Indeed, in the extreme, discovering no resource-based dependencies would be equivalent to baseline SYZKALLER.

Case Study 2: UINPUT. The second case study focuses on UINPUT, a driver for user space input device emulation. For this driver, which makes use of 296 syscalls, the static analysis identifies $94,857$ dependencies, of which 675 remain after filtering.

During our manual false positive evaluation, we identify 225 true positives and 450 false positives. The bottom half of Figure 10 shows the distribution of the dependencies over the weight ranges. Similar to the TCP/IPv4 socket module, *all* false positives are assigned low weights, making it unlikely for SYZGRAPHER to attempt to explore these dependencies.

Our false negative analysis of the UINPUT driver leads to checker candidates that access 34 different resource struct fields in conditionals. Performing the same analysis as for the TCP/IPv4 socket module, we sample 10 fields. Our analysis identified 206 missed dependencies. The root cause (and implications) for the false negatives is the same as for the TCP/IPv4 socket.

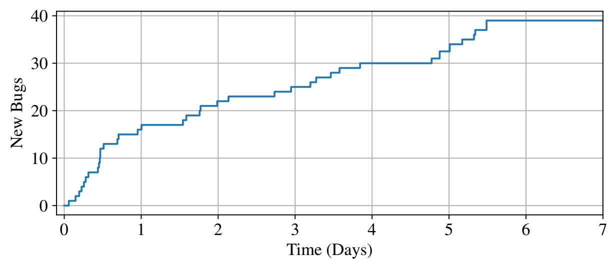


Fig. 11. Trend of new bugs found during the 7-day fuzz campaign.

D. New Bugs (RQ4)

To assess the ability of SYZGRAPHER to find new vulnerabilities, we ran it on 4 versions of the Linux kernel: the latest version (6.9), a stable release (6.8.3), and two long-term support releases (6.6.31, 5.15.159). Each kernel was compiled with KASAN (following the configuration used by SYZBOT) to detect memory-corruption-related errors. It is worth noting that both the latest version (6.9) and several long-term support versions (including 5.15.159) are continuously fuzzed at scale by SYZKALLER.

Over the 7-day period, SYZGRAPHER identified 177 unique bugs across the four versions. The majority were found in the oldest kernel 5.15.159 (91), followed by the more modern versions: 6.6.31 (50), 6.8.3 (55), and 6.9 (42). Some bugs were found in multiple versions; hence the total number of unique bugs is fewer than the sum of these per-version counts.

Of these 177 bugs, SYZGRAPHER found 38 that were previously unknown (i.e., zero-days). While SYZGRAPHER found new bugs in *every version* of the kernel we tested, a majority were found in 5.15.159 (24) despite the fact that this is one of the kernel versions continuously fuzzed by SYZBOT. The remaining bugs were found in 6.6.31 (4), 6.8.3 (6), and 6.9 (3). One new bug was found in three of the kernel versions (all except 6.6.31). In Figure 11, we display the number of new bugs identified over time. Remarkably, 16 of these 38 new bugs were found within the first day of fuzzing.

SYZGRAPHER identified a variety of different types of bugs

Subsystem	Type	Version	Days	Reported
kernel	Deadlock	6.9/6.8.3/ 5.15.159	2.13	✓
fs	Null Pointer	5.15.159	0.19	✓
kernel	Deadlock	5.15.159	0.45	✓
fs	Deadlock	5.15.159	0.31	
block	Kernel Fault	5.15.159	0.28	✓
drivers	Logic	5.15.159	0.06	
fs	Logic	5.15.159	1.00	✓
fs	Hang	5.15.159	0.71	
fs	Hang	5.15.159	1.59	✓
fs	Logic	5.15.159	0.95	✓
kernel	Logic	5.15.159	1.77	✓
mm	Logic	5.15.159	2.73	✓

TABLE I
BUGS FOUND DURING THE 7-DAY FUZZ CAMPAIGN FOR WHICH WE HAVE ISOLATED A REPRODUCER.

including process hangs (11), stalls (6), deadlocks (3), logical errors such as reachable assertions (11), kernel faults (2), and memory corruption such as use-after-free or out-of-bounds accesses (5).

In order to provide actionable feedback to the Linux team, we generate a reproducer for each bug before submitting a report. We successfully generated reproducers for 12 of 38 identified bugs (listed in Table I). Note that our success rate for generating reproducers (32%) is on par with that of SYZKALLER. At the time of writing, there are 23,987 reports on the public dashboard, of which only 7,756 (32%) have reproducers. The failure to generate reproducers is a fundamental problem of kernel fuzzing, not unique to SYZGRAPHER. It is often difficult to generate reproducers for bugs triggered in non-deterministic ways or those which depend on long-term accumulated global state. Of the 12 bugs we were able to extract reproducers for, we have reported all except three, which affected obsolete components, and hence would not receive updates. One of the bugs was already confirmed by the developers. A complete list of bugs found during the campaign is provided in the Appendix D (Table II).

VI. LIMITATIONS

Conditional Dependencies. The dependency model used in our analysis does not consider extra arguments (such as integers) passed to the syscall which may influence the setter/checker. Such an analysis could reveal finer-grained dependencies but would be much more expensive.

Indirect Resource Lookup. Our analysis depends on resources to be explicitly passed as arguments to a syscall; setters and checkers must share the same type. This pattern is common in Linux, but our analysis may miss cases where resources are either deeply nested or associated with some global state such that they don't appear as an argument.

Check Type. We currently do not consider patterns in the checker that use types of checks other than direct value comparisons–for example, bitwise comparisons–as these can be harder to interpret.

VII. RELATED WORK

Relation-learning Kernel Fuzzers. The most similar works to SYZGRAPHER are those which learn *relations* or *dependencies* to generate more interesting test cases during fuzzing. SYZKALLER [6] for example, maintains a *choice table*, which dynamically keeps track of which pairs of syscalls appear frequently together in the corpus and uses this table to sample during mutation. MOONSHINE [3] performs a similar analysis on syscall traces of real-world programs to create a *distilled* corpus that retains these syscall relations. HEALER [2] analyzes minimized test cases in the corpus to learn *influence relations*, similar to SYZKALLER's choice table, and extends the mutation and generation routines to use this information more effectively. IMF [4] performs an analysis on real-world traces to learn both *ordering dependence* and *value dependence* of syscalls. HFL [5] extends this idea further, applying offline static analysis and dynamic symbolic execution to learn (among other things) syscall relations. ACTOR [1] learns a different type of relation based on *actions* such as *alloc*/*free*/*write* and uses this to synthesize new test cases.

In the line of research on syscall specification generation [23], [24], recent work Syzgen++ [25] improves the specification generation via symbolic execution establishing the relationship between resource producers and consumers. As discussed in Section II-A, SYZGRAPHER uses `Syzlang` system call specifications and can be coupled with all these approaches.

While SYZGRAPHER also learns *dependencies*, the key novelty is that these dependencies specify explicit resource linkage (not just syscall order), and our analysis is therefore capable of discerning both *which resources* are involved in a check (Section IV-C) and which *entrypoint* is associated with a given indirect syscall invocation (Section A), in the case where a syscall may dispatch to different internal functions based on user arguments (such as `socket`).

State-based Fuzzing. An alternative approach to improving fuzzer efficacy is to augment fuzzer feedback with more detailed information. STATEFUZZ [9] identifies *state* variables in the kernel and maps their values to new feedback. Similar concepts have been applied in user-space fuzzing [26], [27], [28], [29], [30]. While this indirect approach to finding dependencies is relatively simpler, it can be difficult to implement automatically—indeed, many user-space approaches rely on *manual* annotations. Additionally, in the context of kernel fuzzing where each test-case invocation is relatively expensive, such feedback-driven approaches are not guaranteed to *maintain* a proper dependency during mutation even if they were able to discover one, while SYZGRAPHER that *explicitly* learns dependencies is able to do so.

Extending Kernel Fuzzing. Orthogonal to techniques which learn dependencies (such as SYZGRAPHER), there has also been much research in new ways to extend the general capabilities of kernel fuzzers, through simulation of virtual devices [31], [32], [33], accelerated fuzz-loops with checkpoints [34], modeling the state of virtual devices [35], fuzzing at the hardware-OS boundary [36], leveraging the user space app and randomizing parts of its communication with the kernel [37], replaces parts of the original calls to the kernel from the user-space app with the fuzzer inputs, improving reproducers via distance information [38], and modeling *semantic bugs* [39].

VIII. Conclusion

In this work, we introduced SyzGrapher, a novel resource-centric, graph-based kernel fuzzer designed to enhance the discovery of kernel bugs by automating the construction of semantically correct syscall sequences. By performing a dependency analysis, SyzGrapher statically identifies resource-based dependencies among syscalls and integrates this information into a graph-based mutation engine. This approach allows for the generation of syscall sequences that respect resource handling semantics.

The evaluation of SyzGrapher demonstrated improvements in both code coverage and bug discovery compared to the state-of-the-art kernel fuzzing techniques.

Acknowledgment

This material is based on research sponsored by DARPA under agreement number N66001-22-2-4037. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[1] M. Fleischer, D. Das, P. Bose, W. Bai, K. Lu, M. Payer, C. Kruegel, and G. Vigna, "ACTOR: Action-guided kernel fuzzing," in *Proc. USENIX Security Symposium*, 2023.

[2] H. Sun, Y. Shen, C. Wang, J. Liu, Y. Jiang, T. Chen, and A. Cui, "Healer: Relation learning guided kernel fuzzing," in *Proc. ACM SIGOPS Symposium on Operating Systems Principles*, 2021.

[3] S. Pailoor, A. Aday, and S. Jana, "Moonshine: Optimizing OS fuzzer seed selection with trace distillation," in *Proc. USENIX Security Symposium*, 2018.

[4] H. Han and S. K. Cha, "Imf: Inferred model-based fuzzer," in *Proc. Conference on Computer and Communications Security*, 2017.

[5] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "HFL: hybrid fuzzing on the linux kernel," in *Proc. The Network and Distributed System Security Symposium*, 2020.

[6] Google, "Syzkaller: An unsupervised coverage-guided kernel fuzzer," https://github.com/google/syzkaller, 2017.

[7] H. Green and T. Avgerinos, "Graphfuzz: Library api fuzzing with lifetime-aware dataflow graphs," in *Proc. International Conference on Software Engineering*, 2022.

[8] https://syzkaller.appspot.com/upstream/fixed, accessed: 2024-06-06.

[9] B. Zhao, Z. Li, S. Qin, Z. Ma, M. Yuan, W. Zhu, Z. Tian, and C. Zhang, "StateFuzz: System Call-Based State-Aware linux driver fuzzing," in *Proc. USENIX Security Symposium*, 2022.

[10] Y. Hao, G. Li, X. Zou, W. Chen, S. Zhu, Z. Qian, and A. A. Sani, "Syzdescribe: Principled, automated, static generation of syscall descriptions for kernel drivers," in *Proc. IEEE Symposium on Security and Privacy*, 2023.

[11] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "DIFUZE: Interface aware fuzzing for kernel drivers," in *Proc. Conference on Computer and Communications Security*, 2017.

[12] A. Fioraldi, "Program state abstraction for feedback-driven fuzz testing using likely invariants," 2020.

[13] R. Natella, "Stateafl: Greybox fuzzing for stateful network servers," *Empirical Software Engineering.*, 2022.

[14] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "DR. CHECKER: A soundy analysis for linux kernel drivers," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[15] Q. Wu, A. Pakki, N. Emamdoost, S. McCamant, and K. Lu, "Understanding and detecting disordered error handling with precise function pairing," in *Proc. USENIX Security Symposium*, 2021.

[16] Y. Kang, B. Ray, and S. Jana, "Apex: Automated inference of error specifications for c apis," in *Proc. IEEE/ACM International Conference on Automated Software Engineering*, 2016.

[17] K. Lu, A. Pakki, and Q. Wu, "Detecting Missing-Check bugs via semantic- and Context-Aware criticalness and constraints inferences," in *Proc. USENIX Security Symposium*, 2019.

[18] K. Sparck Jones, "A statistical interpretation of term specificity and its application in retrieval," *Journal of documentation*, vol. 28, 1972.

[19] Google, "Syzbot: Continuous Linux kernel fuzzing," https://syzkaller.appspot.com/upstream, 2017.

[20] "Moonshine's compatibility with newer kernels," https://github.com/shankarapailoor/moonshine/issues/4.

[21] "Cross-pollination," https://github.com/google/fuzzing/blob/master/docs/glossary.md.

[22] "Healer private vs public version," https://github.com/SunHao-0/healer/issues/37#issuecomment-950465293.

[23] H. Sun, Y. Shen, J. Liu, Y. Xu, and Y. Jiang, "{KSG}: Augmenting kernel fuzzing with system call specification generation," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 351–366.

[24] Y. Hao, G. Li, X. Zou, W. Chen, S. Zhu, Z. Qian, and A. A. Sani, "Syzdescribe: Principled, automated, static generation of syscall descriptions for kernel drivers," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 3262–3278.

[25] W. Chen, Y. Hao, Z. Zhang, X. Zou, D. Kirat, S. Mishra, D. Schales, J. Jang, and Z. Qian, "Syzgen++: Dependency inference for augmenting kernel driver fuzzing," in *IEEE Symposium on Security and Privacy*, 2024.

[26] J. Ba, M. Böhme, Z. Mirzamomen, and A. Roychoudhury, "Stateful greybox fuzzing," in *Proc. USENIX Security Symposium*, 2022.

[27] C. Aschermann, S. Schumilo, A. R. Abbasi, and T. Holz, "Ijon: Exploring deep state spaces via fuzzing," in *Proc. IEEE Symposium on Security and Privacy*, 2020.

[28] A. Fioraldi, D. C. D'Elia, and D. Balzarotti, "The use of likely invariants as feedback for fuzzers," in *Proc. USENIX Security Symposium*, 2021.

[29] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: A greybox fuzzer for network protocols," in *Proc. IEEE International Conference on Software Testing, Validation and Verification*, 2020.

[30] R. Padhye, C. Lemieux, K. Sen, L. Simon, and H. Vijayakumar, "Fuzzfactory: domain-specific fuzzing with waypoints," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[31] Z. Ma, B. Zhao, L. Ren, Z. Li, S. Ma, X. Luo, and C. Zhang, "Printfuzz: Fuzzing linux drivers via automated virtual device simulation," in *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022.

[32] H. Peng and M. Payer, "USBFuzz: A framework for fuzzing USB drivers by device emulation," in *Proc. USENIX Security Symposium*, 2020.

[33] Z. Shen, R. Roongta, and B. Dolan-Gavitt, "Drifuzz: Harvesting bugs in device drivers from golden seeds," in *Proc. USENIX Security Symposium*, 2022.

[34] D. Song, F. Hetzelt, J. Kim, B. B. Kang, J.-P. Seifert, and M. Franz, "Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints," in *Proc. USENIX Security Symposium*, 2020.

[35] K. Kim, T. Kim, E. Warraich, B. Lee, K. R. B. Butler, A. Bianchi, and D. J. Tian, "Fuzzusb: Hybrid stateful fuzzing of usb gadget stacks," in *Proc. IEEE Symposium on Security and Privacy*, 2022.

[36] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz, "PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary," in *Proc. The Network and Distributed System Security Symposium*, 2019.

[37] A. Bulekov, B. Das, S. Hajnoczi, and M. Egele, "No grammar, no problem: Towards fuzzing the linux kernel without system-call descriptions," in *Network and Distributed System Security (NDSS) Symposium*, 2023.

[38] X. Tan, Y. Zhang, J. Lu, X. Xiong, Z. Liu, and M. Yang, "Syzdirect: Directed greybox fuzzing for linux kernel," in *Proc. Conference on Computer and Communications Security*, 2023.

[39] S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, and T. Kim, "Finding semantic bugs in file systems with an extensible fuzzing framework," in *Proc. ACM SIGOPS Symposium on Operating Systems Principles*, 2019.

[40] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim, "Precise and scalable detection of double-fetch bugs in OS kernels," in *Proc. IEEE Symposium on Security and Privacy*, 2018.

[41] K. Lu and H. Hu, "Where does it go? refining indirect-call targets with multi-layer type analysis," in *Proc. Conference on Computer and Communications Security*, 2019.

Appendix

A. Indirect Call Target Identification

Indirect call target identification is made more difficult by the way polymorphism is implemented in the kernel: Specifically, the code that ends up being executed (when a syscall is invoked) might depend on the *type* of the resource passed into this syscall as argument. For example, the `read` syscall in `Syzlang` is defined to work on any file descriptor (`fd`). However, invoking `read` on a file descriptor pointing to a device driver will reach different code than invoking `read` on a file descriptor pointing to `STDIN` or invoking `read` on a socket. At the time of invocation, the `read` syscall will dynamically dispatch to the correct function based on the type of the provided resource.

Prior work tried to solve the indirect call target identification challenge through a function-signature-based analysis [40], [15] which internally uses type-based analysis [41]. Such an analysis maps the function implementation to the function pointer according to the static type analysis of the address-taken function parameters and function pointer type. Unfortunately, this approach often leads to over-approximation, which comes with many false positives. Thus, we propose a combination of *static* and *dynamic* analyses for precise indirect call target resolution.

As described above, the indirect dispatch depends on the resource type. That is, even if the indirect call is performed in a resource-consuming syscall, its destination is determined by the corresponding resource's constructor. In particular, the execution of the resource's constructor sets the function pointers deferenced by the resource-consuming syscall. Therefore, to determine the indirect call targets, we call the constructors for each resource before it is used as a syscall argument. Resource constructors usually expect arguments of the string or integer types. Thus, the problem of indirect call target identification requires identifying values used as arguments when executing resource constructors. Some of these values are also provided by `Syzlang` definitions, and some we extract from the kernel build information. For the rest of the (integer-type) arguments, we perform a static dataflow analysis of the constructor arguments and observe if the conditional statements test them. If such a conditional statement is found, we use it as a range-defining statement for the corresponding constructor argument. A similar analysis for string-type arguments is future work.

Once we obtained the argument values required for the resource constructors, we proceeded further with the indirect call target identification phase ❶ by executing each resource's constructor concretely. Then, we inspect kernel memory and supply all the dynamic target resolvents to the subsequent dependency analysis phase. This information allows the dependency analysis to *precisely* determine the targets of indirect calls whenever it encounters them.

Our indirect call analysis is tailored to the specific fuzz target configuration, which enhances its precision. However, this also means that a change in the target configuration (e.g., underlying hardware) may partially invalidate analysis results and require a repeat of the analysis. In this case, it is sufficient to only repeat the analysis for the modified kernel components.

B. Differences between GraphFuzz and SyzGrapher

1) Differences in Requirements: The original GraphFuzz paper proposed graph completion in both the forward direction (generating resource users/destructors) and the reverse direction (generating resource producers/constructors). The result of this bi-directional completion is that all resources are explicitly constructed and then later destructed (no resources are left dangling without a destructor). In the context of in-memory library fuzzing, this choice was critical for avoiding memory leaks. However, in the context of kernel fuzzing, we find that enforcing forward graph completion (requiring every resource to be *explicitly destructed*) is actually disadvantageous. The kernel will automatically clean up any leftover resources after each program invocation, so adding all these destructors simply slows down execution speed and increases the size of the generated programs. Graph-based mutations can still add new resource consumers/destructors, but we don't enforce that all resources need to be explicitly destructed.

2) Differences in Threat Model: The threat model for library API fuzzing (which GraphFuzz was designed for) and kernel fuzzing are quite different. In library API fuzzing, crashes due to violations of the API specification are usually not interesting. For example, passing a deallocated memory object as an argument or a random constant instead of a pointer may cause a library function to crash, but it is unlikely that an attacker would have such control over the inputs. Rather, the assumption is that an attacker can invoke the API in any way that is legal according to the specification.

On the other hand, kernel fuzzing has no such constraints. A malicious user-space program can invoke syscalls with *any* arguments, no matter how malformed. The kernel needs to be resilient against any kind of user-space attacker. Generating *more correct* inputs through graph-based fuzzing is therefore a matter of *fuzzer efficiency* not of *necessity*. In fact, some percent of the time, we *do* want to perform potentially incorrect structural mutations to explore the full state space of the kernel.

To this end, we integrate our new graph-based mutators in conjunction with Syzkaller's existing mutators. Half of the time, we select a graph-based mutator (which is more likely to respect structure) and the other half we use Syzkaller's default mutators (which may destroy structure or introduce invalid types). Interesting inputs (regardless of validity) will be maintained in the corpus and can later be mutated with either set of mutators.

C. Full Test Cases

In Figure 12, Figure 13, and Figure 14, we present the full texts of the programs discussed in Section V-B.

```
r0 = openat$kvm(0xffffffffffffff9c, &(0x7f0000000080),
  0x0, 0x0)
r1 = ioctl$KVM_CREATE_VM(r0, 0xae01, 0x0)
ioctl$KVM_CREATE_IRQCHIP(r1, 0xae60)
r2 = ioctl$KVM_CREATE_VCPU(r1, 0xae41, 0x0)
ioctl$KVM_CAP_HYPERV_SYNIC(r2, 0x4068aea3,
  &(0x7f0000000000))
ioctl$KVM_CREATE_VCPU(r1, 0xae41, 0x0)
ioctl$KVM_RUN(r2, 0xae80, 0x0)
```

Fig. 12. Full test case for Figure 9, program A.

```
r0 = openat$sw_sync(0xffffffffffffff9c,
  &(0x7f0000000080), 0x0, 0x0)
ioctl$SW_SYNC_IOC_CREATE_FENCE(r0, 0xc0285700,
  &(0x7f0000000040)={0x1, "68a711fe1799dcc5cf77d0-
  ba334236db21adb98c55de63b6dd4855c9b7e16d4a",
  <r1=>0x0})
r2 = openat$sw_sync(0xffffffffffffff9c,
  &(0x7f0000000100), 0x0, 0x0)
ioctl$SW_SYNC_IOC_CREATE_FENCE(r2, 0xc0285700,
  &(0x7f00000000c0)={0x6, "b9b971a1fc-
  be4e79b4a32c842ac40ebb7e9a972b7811e4bd3d17954-
  fa0acd797", <r3=>0x0})
ioctl$SYNC_IOC_MERGE(r1, 0xc0303e03,
  &(0x7f0000000000)={"f378b2c7e641211237ad1358f2c00850
  53d782d4e4883a9d2873c43c979c7f90", r3, <r4=>0x0})
dup2(r0, r4)
```

Fig. 13. Full test case for Figure 9, program B.

```
r0 = socket$inet_smc(0x2b, 0x1, 0x0)
bind$inet(r0, &(0x7f0000000040)={0x2, 0x4e22, @multi-
  cast2}, 0x10)
connect$inet(r0, &(0x7f0000000000)={0x2, 0x4e22,
  @dev={0xac, 0x14, 0x14, 0x30}}, 0x10)
r1 = socket$nl_route(0x10, 0x3, 0x0)
r2 = socket$inet6_udp(0xa, 0x2, 0x0)
ioctl$ifreq_SIOCGIFINDEX_batadv_hard(r2, 0x8933,
  &(0x7f0000000480)={'batadv_slave_0\x00', <r3=>0x0})
sendmsg$nl_route(r1, &(0x7f0000000200)={0x0, 0x0,
  &(0x7f00000000c0)={&(0x7f00000006c0)=ANY=[@ANY-
  BLOB="3c000000100001000000000000000000007000000",
  @ANYRES32=r3, @ANY-
  BLOB="000000000000016801800018014"], 0x3c}}, 0x0)
sendto$inet(r0, &(0x7f0000000180)='J', 0x10002, 0x80,
  0x0, 0x0)
```

Fig. 14. Full test case for Figure 9, program C.

D. Results

Table II presents all new bugs found during our 7-day fuzzing campaign.

Subsystem	Operation	Type	Version	Days	Report status
mm	hugetlb_wp	Hang	6.9	0.51	-
fs	ext4_map_blocks	Hang	6.9	0.47	-
kernel	__perf_event_task_sched_in	Deadlock	6.9, 6.8.3, 5.15.159	2.13	Reported
net	addrconf_rs_timer	Stall	6.9	5.01	-
fs	fs_bdev_sync	Hang	6.8.3	1.54	-
kernel	__mod_timer	Stall	6.8.3	5.49	-
net	__netif_receive_skb_core	Stall	6.8.3	5.17	-
net	addrconf_rs_timer	Stall	6.8.3	5.01	-
net	ipv6_list_rcv	Stall	6.8.3	5.32	-
net	rfkill_register	Hang	6.8.3	5.34	-
kernel	console_lock	Logic	6.6.31	0.15	-
kernel	alloc_workqueue	Hang	6.6.31	0.69	-
drivers	fbcon_putcs	Null Pointer	6.6.31	3.46	-
net	addrconf_rs_timer	Logic	6.6.31	4.88	-
fs	xlog_cil_commit	Null Pointer	5.15.159	0.19	Reported
drivers	floppy_queue_rq	Logic	5.15.159	0.46	-
drivers	floppy_shutdown	Logic	5.15.159	0.26	-
fs	add_transaction_credits	Hang	5.15.159	0.44	-
fs	freeze_super	Deadlock	5.15.159	0.45	Reported
fs	gfs2_recover_journal	Hang	5.15.159	0.47	-
fs	reiserfs_ioctl	Deadlock	5.15.159	0.31	Obsolete component
mm	move_to_new_page	UAF Read	5.15.159	0.23	-
block	bio_associate_blkg_from_css	Kernel Fault	5.15.159	0.28	Reported
drivers	__floppy_read_block_0	Logic	5.15.159	006	Obsolete component
fs	fscache_free_cookie	Logic	5.15.159	1.00	Fixed
fs	hfsplus_find_init	Hang	5.15.159	0.71	Obsolete component
fs	ocfs2_get_sector	Hang	5.15.159	1.59	Reported
fs	__brelse	Logic	5.15.159	0.95	Reported
kernel	__perf_event_overflow	Logic	5.15.159	1.77	Reported
drivers	process_fd_request	Logic	5.15.159	1.99	-
drivers	usb_read/usb_submit_urb	Logic	5.15.159	1.76	-
drivers	dvb_usbv2_generic_write	Kernel Fault	5.15.159	3.20	-
fs	__get_node_page	Hang	5.15.159	3.58	-
fs	reiserfs_sync_fs	Hang	5.15.159	2.95	-
drivers	usb_create_ep_devs	UAF Read	5.15.159	3.27	-
mm	get_pat_info	Logic	5.15.159	2.73	Reported
fs	check_igot_inode	OOB Read	5.15.159	3.84	-
net	mrp_join_timer	Stall	5.15.159	4.77	-

TABLE II
NEW BUGS FOUND BY SYZGRAPHER DURING A 7-DAY FUZZ CAMPAIGN. THE STATUS COLUMN SHOWS THE REPORT STATUS OF ALL BUGS WITH A REPRODUCER.