No Grammar, No Problem: Towards Fuzzing the Linux Kernel without System-Call Descriptions

Alexander Bulekov[†*] Bandan Das[*] Stefan Hajnoczi[*] Manuel Egele[†]
alxndr@bu.edu bsd@redhat.com stefanha@redhat.com megele@bu.edu
[†]Boston University [*]Red Hat

***Abstract*—The integrity of the entire computing ecosystem depends on the security of our operating systems (OSes). Unfortunately, due to the scale and complexity of OS code, hundreds of security issues are found in OSes, every year [32]. As such, operating systems have constantly been prime use-cases for applying security-analysis tools. In recent years, fuzz-testing has appeared as the dominant technique for automatically finding security issues in software. As such, fuzzing has been adapted to find thousands of bugs in kernels [14]. However, modern OS fuzzers, such as Syzkaller, rely on precise, extensive, manually-created harnesses and grammars for each interface fuzzed within the kernel. Due to this reliance on grammars, current OS fuzzers are faced with scaling-issues.**

In this paper, we present FUZZNG, our generic approach to fuzzing system-calls on OSes. Unlike Syzkaller, FUZZNG does not require intricate descriptions of system-call interfaces in order to function. Instead FUZZNG leverages fundamental kernel design features in order to reshape and simplify the fuzzer's input-space. As such FUZZNG only requires a small config, for each new target: essentially a list of files and system-call numbers the fuzzer should explore.

We implemented FUZZNG for the Linux kernel. Testing FUZZNG over 10 Linux components with extensive descriptions in Syzkaller showed that, on average, FUZZNG achieves 102.5% of Syzkaller's coverage. FUZZNG found 9 new bugs (5 in components that Syzkaller had already fuzzed extensively, for years). Additionally, FUZZNG's lightweight configs are less than 1.7% the size of Syzkaller's manually-written grammars. Crucially, FUZZNG achieves this without initial seed-inputs, or expert guidance.

I. Introduction

The Operating System continues to serve as one of the most security-critical building blocks in modern computing. The OS' role in managing resources and enforcing isolation between applications makes it a target for attackers who seek to violate OS-provided guarantees. Recognizing the critical nature of OS security, fuzzers have identified and helped fix thousands of bugs in OS kernels. Recently, the success of OS fuzzers has emphasized difficulty of writing secure low-level code, and has even spurred initiatives such as support for safer languages in the Linux kernel, and the usage of hardware-features such as Memory Tagging to enable advanced low-overhead defenses against memory-corruption issues [23], [44]. Most OS fuzzers focus on the critical *system-call* interface, which enables user-space applications to request services from the kernel.

Syzkaller[14], the most prolific system-call fuzzer, has become an integral component of the Linux Kernel development lifecycle, with over 2,700 mentions in kernel commit messages. As such, syzkaller, itself, has grown to be a sizeable project, with over 200 contributors. Crucially, Syzkaller can only fuzz system-calls that are sufficiently described by a "syzlang" grammar. These grammars encode and annotate the types of resources provided as inputs and returned as outputs, by system-calls. Therefore, much of the syzkaller community's work is focused around developing and refining "syzlang" descriptions for system-calls, which are essential to Syzkaller's success.

Developing such grammars is a manual process, and requires detailed knowledge about the interface (i.e., set of system calls) in question. As such, grammars are prone to human-error, and can lead to gaps in coverage, or over-fitting (preventing the fuzzer from exploring all states and scenarios in which code could be covered). Additionally, syzkaller sometimes requires writing supplementary harnessing code to fuzz particularly complex interfaces. For example, to fuzz the Linux Kernel Virtual Machine (KVM) interface, which powers security-critical virtualization software, Syzkaller developers committed 891 lines of detailed syscall descriptions, 243 KVM-related constants, and a further 879 lines of KVM-specific C harnessing code (illustrated in Figure 1). Even though Syzkaller features tens-of-thousands of hand-crafted "syzlang" rules, the current process cannot scale to fuzz the millions of lines of code added to the Linux Kernel each year [33]. Academic works have recognized Syzkaller's scalability problem with manually-created syzlang grammars, and have focused on automatically generating grammars. Works such as Difuze, IMF, SyzGen, and KSG apply static and dynamic-analysis techniques to automatically generate system-call descriptions [12], [18], [9], [51]. Difuze, IMF and SyzGen are designed and evaluated against interfaces, such as Android Drivers, and macOS APIs, for which no baseline manual-descriptions exist. KSG's descriptions appear to improve Syzkaller's coverage, however source-code has not been released and upstream Syzkaller-based Linux fuzzing efforts continue to rely only on manually-written descriptions. Some upstream efforts for description generation are limited to identifying the types of struct arguments passed to ioctl system-calls [36]. Importantly, the Syzkaller project has been tracking the need for automatic generation of Linux system-call descriptions as an open issue, since 2018 [53].

Network and Distributed System Security (NDSS) Symposium 2023
27 February - 3 March 2023, San Diego, CA, USA
ISBN 1-891562-83-5
https://dx.doi.org/10.14722/ndss.2023.24688
www.ndss-symposium.org

```
files = "/dev/kvm", O_RDWR
ioctl[-1, -1, -1]
mmap[0, 0xF000, PROT_READ | PROT_WRITE, MAP_SHARED|MAP_POPULATE,
0xFFFF, -1]
close[-1]
fstat[-1, -1]
read[-1, -1, 0xFFFF]
write[-1, -1, 0xFFFF]
```

Fig. 1: *Top:* Syzkaller's 2013 lines of KVM-specific Descriptions and Harnesses. *Bottom:* FUZZNG config for fuzzing KVM

Other kernel-fuzzing work focuses on improving syzkaller's input-generation [52], [57], improving fuzzing support for complex kernel interfaces [59], fuzzing specific device-kernel interfaces [40], [49], improving the performance of kernel-fuzzing using snapshots [50], developing tools for coverage-guided gray-box fuzzing of OS kernels [47], or combining fuzzing with symbolic-execution to find more bugs [27]. Moonshine[38] has identified the expense of maintaining accurate syscall-descriptions. However, the alternatives proposed require collecting and analyzing extensive system-call traces from real-world programs. Finding and exercising programs that interact extensively with the complex kernel-interfaces is, itself, a complex problem with scalability issues. Fuzzers based on manually-written descriptions or real-world traces can quickly reach deep parts of the kernel code. However, the process of manually collecting system-call traces, or writing descriptions is prone to human-error: the immediate coverage gains from descriptions and traces risk obscuring parts of the code that the fuzzer can no longer exercise meaningfully. As testament to this, FUZZNG found bugs in code that Syzkaller marked as covered.

In this paper, we present FUZZNG, our system for fuzzing system-calls without detailed descriptions of each interface. By default, the kernel exposes an enormous input-space through the system-call interface, encompassing all of a process' virtual-memory (e.g., via pointers to buffers), and the entire file-descriptor table. Thus, even though, on the surface, system-calls accept up to six numerical arguments, a trivial fuzzing strategy that simply passes random arguments to system-calls will not achieve adequate results, since most arguments will be semantically invalid (e.g. pointers to unmapped memory, or non-existent file-descriptor numbers). Current system-call fuzzers rely on detailed system-call descriptions and real-world traces to generate valid system-call arguments, thereby avoiding fuzzing the entire input space associated with process-memory and file-descriptors, provided to user-space applications by the kernel. However, these grammar-based approaches simply shift much of the burden of generating valid system-calls onto the developer.

FUZZNG's core insight is that instead of relying on extensive system-call descriptions, the system-call input space can be *reshaped*, to (1) obviate the need for system-call specific grammars, and (2) make system-call fuzzing conducive to fuzzing with battle-tested off-the-shelf fuzzing tools. To realize this reshaping, FUZZNG leverages the very APIs kernel code already uses to handle system-calls in normal operation – specifically APIs to access user-space memory and manage file-descriptors. By applying input-space reshaping, FUZZNG essentially relies on a variation of the aforementioned "trivial" fuzzing strategy, passing system-call arguments directly from a fuzzer (e.g., *libFuzzer*). Through this technique, FUZZNG eliminates the need for detailed descriptions and harnesses for individual system-calls. In short, FUZZNG does not rely on any detailed preliminary analysis of each system-call, prior to fuzzing.

We implemented FUZZNG for the Linux Kernel, and evaluated it for 10 Linux interfaces for which Syzkaller features extensive descriptions. We found that even though input-space reshaping allows FUZZNG to avoid detailed system-call descriptions, FUZZNG achieves competitive coverage, and significant bug-finding capabilities when compared with Syzkaller. For example, FUZZNG is able to set up a KVM-based VM, configure it with virtual-memory slots, fill it with instructions, run it, causing an exit into KVM instruction-emulation code, and reach a bug, with no special knowledge of the KVM interface, or its 100+ distinct ioctl commands. Similarly, FUZZNG creates inputs that automatically create and execute complex bpf programs, and io_uring sequences. All prior systems that are capable of targeting these complex interfaces, require human-written system-call descriptions. Furthermore, FUZZNG found new issues (detailed in Section V-D) in code that was already covered by Syzkaller, highlighting the fact that manually-written descriptions can often underfit or overfit the interface in question. In summary, we make the following contributions:

- We describe the main features of the input-space exposed by the Linux system-call interface. Then, we cover the system-call grammars common among current fuzzers. Finally, we describe the subtle pitfalls of these manually-written grammars.
- We present FUZZNG, our system-call fuzzer, which obviates the need for expert-curated grammars. Unlike previous approaches, FUZZNG *reshapes* the kernel's input space, rather than attempting to automatically generate system-call descriptions. We demonstrate that FUZZNG is

competitive, when compared with Syzkaller, achieving an average of 102.5% of Syzkaller's coverage over components for which Syzkaller has extensive descriptions. Our results show the pitfalls of grammar-based approaches, by finding bugs in code that Syzkaller already covered.
- We describe FUZZNG's snapshot-based fuzzing approach along with the mutation and execution strategies that we tailored to the problem of grammarless kernel-fuzzing. We present FUZZNG's novel approach to "refining" inputs, leveraging information learned as they are executed, to improve the quality of the input-corpus.
- We report on the 9 new bugs discovered by FUZZNG, of which 5 are in components covered by Syzkaller. To enable further research into grammarless kernel fuzzing, we will open source all of FUZZNG's components.

In the spirit of open science, we will publish the FuzzNG source code at https://github.com/BUseclab/FuzzNG.

II. BACKGROUND

In this section, as background for our work, we describe relevant aspects of operating-systems, system-calls, and OS-Fuzzers. On Linux, system-calls are identified by an integer "id". System-calls support up to six word-sized arguments (communicated through registers on most architectures). System-calls can return a value, which is also provided through a register. As the numeric arguments passed to a system-call are limited in size, some system-call arguments semantically represent larger data-structures. Specifically, the linux kernel-documentation highlights the two types of system-call arguments that are used to indirectly pass arbitrary-sized data to the kernel: *pointers* and *file-descriptors* [1], [2]. System-calls use file-descriptor arguments to allow "userspace to refer to a kernel object"[1]. For system-calls that involve a large number of arguments, the arguments are placed "into a structure that is passed in [to the kernel] by pointer"[2]. For example, `alarm` schedules the delivery of a signal to the process. The only argument – the number of seconds until the alarm expires – is passed to the kernel, directly, as an integer. However, the `write` system-call expects three arguments: a file-descriptor, a buffer that will be written to the file, and the length of the buffer. While the file-descriptor and the length of the buffer are represented as integers that can fit into registers, the buffer can have arbitrary length. As such, the kernel expects the register for the buffer argument to *contain the address* of the actual buffer.

Linux has hundreds of system-calls that serve as the main mechanism by which user-space applications can request services from the kernel. Internally however, the kernel can feature many implementations for the same system-call. For example, the `ioctl` system-call, which is used to control devices, serves as main configuration-mechanism for many drivers in the kernel. Applications obtain references (i.e., file descriptors) to drivers, by opening so-called special files associated with the devices (i.e., files commonly located in the `/dev` directory). Then, applications can configure the drivers, by invoking `ioctl`, with the file-descriptor as the first argument. Internally, the kernel associates each file-descriptor with a `file_operations` struct, which points to the functions that will be called transparently for operations such as `read`-ing, `write`-ing and `close`-ing the file. Using `file_operations`, the kernel routes the `ioctl` request to a device-specific implementation of the system-call, depending on the type of special file (e.g., character device vs. graphics accelerator). As such, due to file-descriptor and pointer arguments, the hundreds of Linux system-calls are simply a narrow window to tens of millions of lines of kernel code.

A. System-call fuzzers and grammars

System-call fuzzers such as Syzkaller and Trinity rely on grammars or rules that aid them in generating valid system-calls and associated file-descriptor and pointer arguments. A grammarless fuzzer could simply pass fuzzer-provided integers as arguments to system-calls. However, such a fuzzer is quickly rendered useless by the effectively boundless system-call input-space induced by pointer and file-descriptor arguments, as discussed in Section I. As such, current system-call fuzzers rely on extensive grammars for kernel interfaces. For example, Syzkaller requires annotations of struct types, flag fields, enums and constants passed as system-call arguments. Furthermore, Syzkaller relies on manual annotations of resources created by each system-call (such as file-descriptors) to determine valid system-call sequences. These annotations will prevent Syzkaller from trying to use the `ioctl` system-call to interact with a driver, without a prior call to `open` to open the corresponding file in `/dev/`.

Even though Linux features several hundred system-calls, system-call behavior can depend drastically on the arguments. For example, the `write` system-call uses vastly different code-paths depending on the type of file written to (e.g., a file on disk vs. a socket or a pipe). Each of the different types of system-call invocations requires its own Syzkaller description. As such Syzkaller features tens of thousands of lines of descriptions, contributed by dozens of developers. These descriptions are written in a domain-specific language called "syzlang", designed to describe system-call interfaces. Despite this gargantuan community effort, the lack of complete system-call descriptions for kernel interfaces is a known limiting factor for current system-call fuzzers [53].

In the remainder of this section, we will describe the main difficulties for system-call fuzzers, and how the grammars used by existing fuzzers attempt to alleviate them.

1) Pointers: As mentioned above, system-calls often expect one or more arguments to contain pointers to data-structures located in the user-space process' memory. Effective fuzzers, such as Syzkaller, feature descriptions of these data-structures, allowing them to identify pointer arguments and to create and mutate the corresponding data-structures. However, pointers create a challenge for naive fuzzers that simply pass mutated values as arguments; even if the value points to a valid virtual address, the naive fuzzer has no knowledge that it should place mutated data at the corresponding location in memory.

Through grammars, Syzkaller is aware when a system-call expects a pointer argument. The grammars encode the length and type of data that the pointer should reference (e.g. a flat buffer, or a struct with individual fields). For example, Figure 2, shows a Syzkaller description for the `VHOST_SET_VRING_ADDR` `ioctl` call (related to the vhost VIRTIO offload subsystem). The final argument to the system-call is indicated as a "pointer" to a `vhost_vring_addr`-

```
1  ioctl$VHOST_SET_VRING_ADDR(fd fd_vhost,
2                             cmd const[VHOST_SET_VRING_ADDR],
3                             arg ptr[in, vhost_vring_addr])
4
5  VHOST_SET_VRING_ADDR = 1076408081
6
7  vhost_vring_addr {
8   index flags[vhost_vring_index, int32]
9   flags int32[0:1]
10  desc_user_addr ptr64[out, array[int8]]
11  used_user_addr ptr64[out, array[int8]]
12  avail_user_addr ptr64[out, array[int8]]
13  log_guest_addrs flags[kvm_guest_addrs, int64]
14 }
```

Fig. 2: Partial example of a Syzkaller description for a *vhost* `ioctl`

type struct. Description-based fuzzers, such as Syzkaller, require detailed annotations of pointer arguments, and *every* struct that the kernel accesses in a user-space process.

2) File-Descriptors: On Linux, files are privileged resources, managed by the kernel. As such, file-operations are initiated through system-calls. First, a user-space process opens a file using a system-call, such as `open`, `socket`, or `pipe`. Internally, the kernel keeps a table of opened files for each process. Since a process usually has multiple files open, the `open` system-call returns an integer *file-descriptor* (or fd) that the process can treat as a handle for the opened-file. For each new opened file, the kernel assigns the lowest available integer file-descriptor, starting with 0. For subsequent file-related system-calls, such as `read`, `write`, `mmap`, and `ioctl`, the process passes the integer file-descriptor as an argument to the kernel. Internally, the kernel uses the `fdget()` API to look-up the corresponding file-object in the process' file-descriptor table, which contains all of the information necessary to properly handle the system-call.

The integer file-descriptors pose a challenge for fuzzers. A typical process only has a handful of files open. As such, the random mutations of a fuzzer are highly unlikely to generate an integer system-call argument that is associated with a valid (i.e., open) file. As previously discussed, the problem is exacerbated by the fact that the behavior of file-related system-calls is highly dependent on the type of file. Even if the fuzzer guesses a valid file-descriptor integer, it would simultaneously need to pick a system-call(and arguments) that are valid for that type of file. To address these problems, the grammars used by syscall fuzzers, rely on special annotations for file-descriptors. For example, in Figure 2 (line 1), the syzkaller description indicates that the first argument to a `VHOST_SET_VRING_ADDR` `ioctl` call must be a file-descriptor. The description also specifies that the descriptor must be of an `fd_vhost` type. Elsewhere, Syzkaller features descriptions for `open("/dev/vhost..")` system-calls whose return values are annotated with the `fd_vhost` type. With such rich descriptions, Syzkaller ensures that it passes valid file-descriptor numbers to the `VHOST_SET_VRING_ADDR` `ioctl` call, and that the file-descriptors are associated with a vhost file that was previously opened.

Thus, file-descriptor and pointer arguments are associated with the large abstract input-spaces (user-space memory and kernel-objects). While system-calls rely on other seemingly-complex types of arguments, such as "magic integers" (where only a few specific values are valid), and flags-fields (where a single integer can represent multiple states/settings), fuzzers have gained powerful mechanisms (e.g. redqueen/cmplog[5] and value-profile[48]) to effectively identify magic-values and fuzz flags passed through integers, as these challenges are not unique to OS kernels. However, fuzzing pointers and file-descriptors fundamentally requires consideration of the corresponding larger input-spaces.

a) Managing File-Descriptors: Linux implements a set of system-calls that are used to manage fds. As mentioned above, system-calls such as `open` and `socket` create new fds. The `close` system-call is used to destroy fds. Additionally, the `dup` family of system-calls can be used to make copies of file-descriptors. For example, the `dup2` system-call, allows a process to duplicate an existing reference to a file onto some specific integer file-descriptor:

```
int dup2(int oldfd, int newfd);
```

After a dup2 call, the file associated with the `oldfd` file-descriptor, is also associated with the `newfd` file-descriptor. In this case, the kernel will associate the file with the user-space-provided file-descriptor number, rather than following the default "lowest-available" strategy, mentioned above.

B. Pointer APIs in Linux

On modern Linux-systems, kernel and user-space memory is placed in separate "halves" of virtual memory. As mentioned above, the kernel frequently interprets system-call arguments as pointers. However, the kernel must treat these pointers with care, as they could potentially originate from a malicious process. A malicious pointer could point to a kernel-address, rather than a user-address, or it could point to a location being written to by another thread (potentially creating a data-race). As accessing data in user-space memory is a common pattern, the Linux kernel implements special APIs, such as `copy_{from,to}_user` and `get_user` that must be used to access data in user-space memory. User-space accesses are treated with such care, that CPU architectures have implemented dedicated facilities to ensure that accidental user-space pointer dereferences are impossible: For Linux running on modern Intel architectures with the *Supervisor Mode Access Prevention(SMAP)* feature, a special bit in the `CR4` register must be cleared for user-space memory access to be possible. This further ensures that access to user-space memory cannot be accidental and should occur through centralized APIs.

C. Automatic Grammar Generation

Multiple works have tried to automatically, or semi-automatically harvest system-call grammars. Grammars can be harvested automatically leveraging static or dynamic-analysis techniques. Often these approaches rely on seed-traces of kernel-interactions, which are difficult to collect, due to a lack of user-space applications that achieve complete coverage over kernel-components. Whitebox static and dynamic analyses focus on relatively-shallow heuristics, such as source-code patterns used to define system-calls and specific ioctl parameter types. Automatically-inferred grammars are disadvantaged by the fact that a small lapse in the analysis can have severe effects on the resulting grammar. For example, if an ioctl relies on multiple nested structs (i.e., a struct pointing to another struct, etc), but the analysis fails to associate a second-level struct with a pointer field in the first-level struct, the grammar will not be able to describe any of layers beyond the second. We will demonstrate in the following sections that FUZZNG does not have the same limitation, since rather than attempting to annotate individual arguments and struct-fields, FUZZNG reshapes and fuzzes the fundamental input-spaces exposed by the kernel to user-space.

III. APPROACH

In this section, we present the features of FUZZNG that enable effective, system-call fuzzing without detailed annotations. FUZZNG reshapes the input-space by hooking into strategic kernel APIs related to pointers and files and, on demand, morphing the semi-random arguments obtained from an off-the-shelf fuzzer into valid values for each system call. In Section IV, we will explain how our solutions fit into the overall design of FUZZNG.

A. Reshaping the input-space

At the core of FUZZNG's approach lies the notion of "reshaping" the system-call input-space. A fuzzer that provides random arguments to system-calls performs poorly, as it operates over an incomplete model of the input-space (it has no mechanism to meaningfully fuzz fd and pointer-based arguments). However, simply extending the fuzzer-accessible input space by e.g., allowing the fuzzer to write fuzzing data to arbitrary locations in process-memory does not improve performance, since the fuzzer is unlikely to guess the user-space addresses that a system-call's implementation will read from. [1] Without feedback, fuzzers that simply pass random integer arguments to system-calls have no way of knowing where to place data for pointer reads, or which file-descriptors should be opened and passed to subsequent system-calls. Reshaping denotes a mechanism, by which FUZZNG provides a dynamic "view" of the *engaged* input-space to the fuzzer. That is, at any moment, FUZZNG is aware of the memory and fds that are actively accessed by the kernel. Using API hooks, FUZZNG pauses execution of a fuzzer-input when the kernel references a pointer or file-descriptor, and takes action to populate the corresponding user-space region or file-descriptor number, prior to resuming input execution. Thus the fuzzer has no need for detailed descriptions of fd or pointer arguments - as FUZZNG intercepts the associated kernel accesses *on-demand*.

One mechanism to realize input-space reshaping could directly replace the return values of the memory-access and file-descriptor APIs with fuzzed data. However, instead, FUZZNG places fuzzed data at the *locations referenced* by the memory-access and file-descriptor APIs. This technique has the advantage of avoiding behaviors that would be impossible in an off-the-shelf kernel without hooks. For example, this approach prevents FUZZNG from providing fuzzed-data at bad pointer-addresses (e.g., null-pointers, or kernel pointers). A significant advantage of this design is that when FUZZNG finds a bug, it is straightforward to generate a "reproducer" that can be provided alongside a bug-report to trigger the bug in an unmodified kernel (independent of FUZZNG). To this end, after fuzzing, FUZZNG converts crashing inputs into simple sequences of system-calls (similar to Syzkaller), eliding the hooks.

[1]Storing fuzzed data at predetermined locations and passing pointers to these locations in system-call arguments as implemented by HyperCube's [46] handling of DMA accesses requires expert insight in the form of comparatively low-fidelity grammars.

By reshaping the input-space, through kernel API hooks, FUZZNG makes the system-call interface tractable for fuzzing, without detailed system-call descriptions, or seed-inputs. Due to the dynamic feedback provided through the API hooks, FUZZNG automatically gains detailed awareness of pointer and file-descriptor arguments that other fuzzers encode in system-call descriptions, offline. Because FUZZNG reshapes' the kernel's fundamental fd and memory input-spaces, its methods work even for complex interfaces, such as KVM and io_uring, where automated techniques such as static analysis of ioctl argument types produce grammars at insufficient detail.

Figure 3 demonstrates that FUZZNG reshapes the input-space, so that the kernel does not reject the majority of inputs due to invalid pointers and fds. Throughout the rest of this section, we will describe FUZZNG's approach for both of these featues of the system-call input-space.

B. Pointers

As mentioned previously, system-calls often rely on pointers provided by user-space applications. The kernel initiates pointer accesses after a system-call has been submitted, and the size of the access depends on the type of and arguments to the system-call. As developers of regular user-space applications have prior expert knowledge about which arguments system-calls treat as pointers (e.g., from documentation and man pages), they write code that populates pointers with corresponding data, prior to invoking the system-call. However, FUZZNG has no knowledge about how the kernel treats individual system-call arguments. Instead, FUZZNG leverages the fact that, in general, accesses from kernel-code to user-space memory must occur through centralized APIs, due to

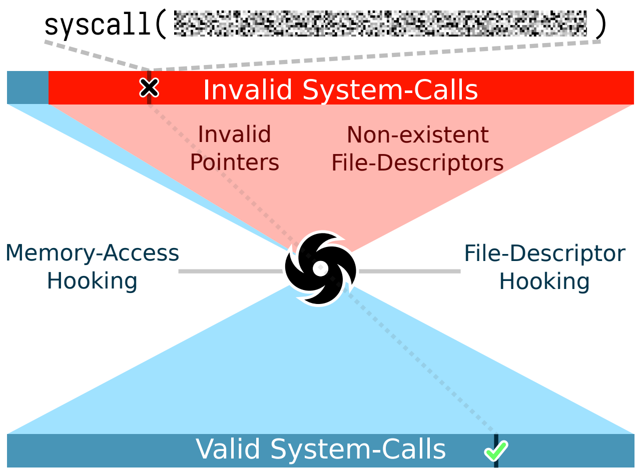

Fig. 3: By default, randomly-generated system-calls cannot reach deep kernel-code paths, as the arguments quickly trigger error conditions. FUZZNG's hooks effectively "reshape" the input-space, transforming nonsensical pointer and file-descriptor arguments into valid ones, without introducing impossible behaviors. Through FUZZNG's kernel hooks, system-calls that would normally result in trivial error conditions, produce interesting behaviors. In this figure and Figure 4, the "warp-hole" symbol represents the parts of FUZZNG that reshape the input-space.

security-measures such as SMAP. For example, the kernel provides a `copy_from_user` function, with similar semantics to `memcpy`, but tailored to allow copying data from user-space memory to kernel-memory. FUZZNG applies lightweight hooks to the copy_from_user and get_user APIs to gain on-demand insight into user-space memory accesses. When a kernel component performs an access using one of these APIs, FUZZNG pauses the access, and fills the corresponding range of user-space memory with fuzzer-provided data. Thus, once FUZZNG resumes the access, the kernel component naturally accesses fuzzer-controlled data. By populating user-memory accesses on-demand, FUZZNG avoids the guess-work associated with identifying locations in process memory that should contain fuzzer-generated data.

User-space access safety net: The vast majority of kernel accesses to user-space memory occur through centralized kernel APIs. However, in rare cases, kernel components may implement their own versions of the APIs, or disable protections. For example, the Kernel Virtual Machine (KVM) subsystem expects VM memory to be mapped in user-space. When a KVM virtual CPU is started (using the `KVM_RUN ioctl`), the kernel directly instructs the CPU to execute instructions from the VM's memory in user-space. In this case, the VM instructions are executed in non-root mode, and the user-space memory assigned to the VM is accessed without SMAP protections. Such behavior would circumvent FUZZNG's reshaping efforts and must thus be handled appropriately. To catch user-space accesses that side-step centralized APIs, FUZZNG employs an additional mechanism to hook user-space memory accesses. To this end, FUZZNG "inflates" the fuzzing process' address space by mapping as many pages as possible (Note that these pages do not consume physical memory, since the OS only allocates physical-pages, when the page is first accessed). By mapping as many virtual-pages as possible, we ensure that accesses to user-space memory are highly likely to correspond to a valid virtual page-mapping. However, these user-space pages are marked as "not present", triggering a page-fault upon any access. Then, using *userfaultfd* (a facility through which page-faults can be handled in userspace), FUZZNG fills each accessed page with fuzzed data, prior to continuing execution of the kernel component. Unlike the API-hooking approach, the userfaultfd-based hooking does not rely on any kernel-code patterns, such as the use of copy_from_user. It thus catches all situations where the kernel tries to access user-space memory without going through the centralized APIs. By default this hook is triggered only once for each page as, for subsequent accesses, the page is marked present, and will not trigger another page-fault. However, as FUZZNG fills the entire page, further reads from the same page will continue to return fuzzed data.

C. File-Descriptors

As discussed in Section II, the kernel often expects system-call arguments to contain file-descriptor numbers. However, a generic fuzzer will provide random numbers for the corresponding arguments, which are highly unlikely to refer to valid file-objects in the kernel. To overcome this fuzzing road-block, FUZZNG associates fuzzer-provided file-descriptor numbers with existing file-objects opened by the process. FUZZNG hooks into the file-descriptor API to ensure that fuzzer-generated fd numbers are reshaped to valid file objects. Internally, FUZZNG leverages the fact that the kernel associates file-descriptors with underlying `struct file` objects, which contain information about the file permissions, offset, supported operations, etc. To resolve file-descriptor numbers to the underlying `struct file` object, kernel developers must use the, `fdget()` API. As such, FUZZNG can hook every attempt to resolve a file-descriptor (valid or invalid). Then, using the `dup2` system-call, FUZZNG can associate the fuzzer-provided (likely invalid) integers with valid files. This ensures that any fuzzer-provided value that is interpreted as a fd by the kernel, is mapped to a valid open file.

IV. FUZZNG

At its core, FUZZNG hooks the fundamental kernel APIs related to system-calls. In this section we describe how these hooks fit into the components comprising the FUZZNG system. First, we describe FUZZNG's fuzzing engine, ① *QEMU-Fuzz*. QEMU-Fuzz is primarily responsible for generating inputs, interpreting coverage data to identify "interesting" inputs, and resetting state in between input executions, via VM-snapshotting. Then, we detail ② *mod-NG*, the modifications made to the Linux kernel to assist with fuzzing. mod-NG hooks the fd and user-memory-access APIs, and error-reporting functions that are used to detect kernel-crashes. Finally, we describe ③ NG-Agent, the user-space agent used to invoke the fuzzed system-calls. NG-Agent is responsible for providing the input to the kernel-under-test, configuring kcov, to collect coverage data over the kernel, and outputting a "canonical" version of each executed input (explained in Section IV-C3c). Figure 4 provides an overview of FUZZNG.

A. QEMU-Fuzz

FUZZNG features QEMU-Fuzz, our VM-snapshot-based fuzzer (① in Fig. 4) built on modified versions of the QEMU-KVM hypervisor, and the *libFuzzer* input mutator. System-calls often lead to modified registers/memory in user-space. They can also establish state within the kernel, which persists across system-calls (e.g. file-descriptor offsets, modified by read/write/seek system-calls). This raises a challenge for fuzzers, which perform best when inputs are executed from an identical starting state. Furthermore, fuzzer-inputs can timeout, or corrupt and crash the NG-Agent process. To address this, FUZZNG executes system-calls against a kernel-under-test running in a VM, and restores the entire VM's state from a snapshot, after each fuzz-input. Similar to Syzkaller, each FUZZNG fuzz-input represents a sequence of system-calls. FUZZNG's snapshot fuzzer ensures that the agent-process and kernel state is completely reset to a consistent snapshot after each input. This approach differs from fuzzers, such as Syzkaller which fuzz the kernel using a "fork-server" to execute inputs. That is, each input is executed in a separate process. Unlike Syzkaller, FUZZNG has no system-call descriptions that it can use to ensure that inputs are well-behaved, and do not create performance problems in the VM.

QEMU-Fuzz implements a fuzzing-specific virtual-device that the VM can use to initialize snapshots and request resets. Additionally the interfaces provided by the virtual-device are used to establish the memory regions where NG-Agent (see § IV-C) running in the VM expects to receive new fuzzer inputs, and the pages where the VM stores kernel-coverage

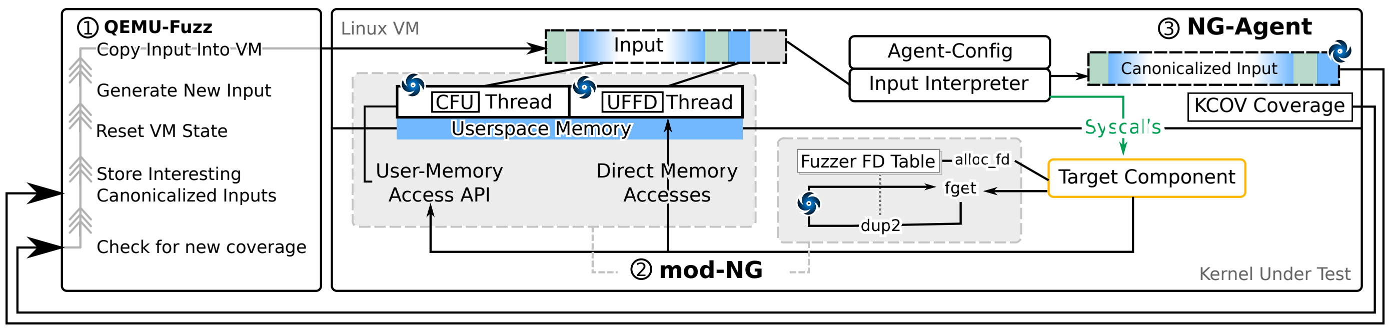

Fig. 4: Overview of our FUZZNG implementation, showcasing the three major components that make up FUZZNG: ① *QEMU-Fuzz* is the snapshot fuzzer that is responsible for generating inputs and resetting VM state after each run. ② *mod-NG* is our collection of kernel-modifications that hook into the user-memory-access and file-descriptor subsystems. ③ *NG-Agent* is the user-space process used to initiate fuzzing system-calls and populate process memory that the process accesses.

data. QEMU-Fuzz places each new input into the agreed-upon location in the guest's physical memory. When the agent has executed the input, it uses the virtual-device interface to request a reset from QEMU-Fuzz. QEMU-Fuzz provides the coverage data in the VM's memory to *libFuzzer* for input mutation. To facilitate this, we modified *libFuzzer* to support the two kcov coverage data formats (program-counter traces and comparison traces). Then, QEMU-Fuzz resets the guest's memory, register and device state, to a previously-initialized snapshot. QEMU's built-in VM snapshot/reload functionality is optimized for long-term snapshot storage, or live-migration, rather than fuzzing. To tailor to fuzzing-specific snapshot workloads, QEMU-Fuzz implements custom VM resets which store all snapshots in-memory, and use KVM's dirty-page tracking to reset only the pages that were written to, since the last snapshot, reducing overhead. Finally QEMU-Fuzz uses a timer to force the VM to reset if an input's execution exceeds the configured timeout.

B. *mod-NG*

In Section III we explain that FUZZNG hooks kernel-code related to user-memory accesses and file-descriptors. To do this, we added a kernel-module, mod-NG(② in Fig. 4), which contains the code that we use to intercept access through `copy_from_user`-type APIs, and file-descriptor operations.

1) copy_from_user APIs: When mod-NG intercepts a user-memory read API call, it wakes-up a userspace "copy-from-user" handler thread (`CFU` in Figure 4) and provides the thread with details about the read location and size. The `CFU` thread is tasked with filling the corresponding location in userspace-memory with fuzzed data. Once the `CFU` thread notifies the kernel that it has handled the request, mod-NG resumes the `copy_from_user` call. Note that we use a combination of the `CFU` thread and mod-NG, rather than simply modifying `copy_from_user` to directly copy fuzzer-input bytes into kernel-memory, for multiple reasons.

1) By filling the read in userspace, we ensure that the process can actually write to the region. Thus, we avoid providing data at impossible locations (e.g., if the read targets unmapped memory).
2) mod-NG does not need to be aware of the fuzzer input. The only component in the VM that directly interacts with the fuzzer input is NG-Agent.
3) The userfaultfd safety net mechanism (`UFFD` in Fig. 4) for hooking user-memory accesses (see Section III-B) is also based on a user-space thread. Handling both `CFU` and userfaultfd hooks in similar userspace threads ensures design-consistency: only user-space components read directly from the fuzz-input. Similar to an unmodified system, system-call inputs (including memory buffers) are populated in the context of the user-space process.

2) File-Descriptor APIs: mod-NG reshapes the input-space introduced by numerical file-descriptors. That is mod-NG ensures that arbitrary fuzzer-provided fd numbers are associated with actual underlying file objects in the kernel. Otherwise, the kernel would reject the majority of system-calls, as they would contain references to unallocated fd-numbers.

To reshape the fd space, mod-NG hooks the `alloc_fd` API which is used by the kernel to allocate new file-descriptors, to handle system-calls, such as `open`. By hooking `alloc_fd`, mod-NG keeps track of all of the fd numbers allocated by the agent process on an "FD stack".

When the kernel uses the `fdget` API to obtain the underlying file object associated with an fd number, mod-NG checks the stack to determine whether the fd number is already allocated. If the fd is allocated, mod-NG simply resumes the API's execution, returning the existing file-object, associated with the fd. However, if the fd is not allocated, mod-NG invokes `dup2` in order to duplicate an existing fd (taken from the fd stack) onto the fd number that was passed as the argument to `fdget`. By default, mod-NG duplicates the most-recently allocated fd (at the top of the stack). However mod-NG also exposes a special system-call, `fuzz-set-fd-offset`, to NG-Agent, which is used to select the index from the top of the fd stack that is passed to the subsequent `dup2` operation. We describe how this system-call is invoked in further detail in Section IV-C3b.

3) Crash Hooking: mod-NG hooks the kernel's `dump_stack` API, which is invoked when a kernel error occurs. In this case, mod-NG uses Port-IO to notify QEMU-Fuzz that the input should be saved as a potential crash, for analysis. mod-NG also tracks NG-Agent process crashes (`SEGFAULTs`) and uses Port-IO to request an immediate VM restore from QEMU-Fuzz. The agent process can crash, as there is no defense preventing the fuzzed system-calls from overwriting sensitive code, heap, etc. regions in the user-space process. Agent crashes do not indicate problems in the kernel, but they slow down the fuzzing process, as the Agent is not able to notify QEMU-Fuzz that the input finished executing (leading to timeouts). To avoid timeouts in cases where NG-Agent crashes, the `SEGFAULT` tracking allows FUZZNG, to improve fuzzing performance.

C. NG-Agent

As FUZZNG is a system-call fuzzer, and system-calls are initiated by user-space applications, we rely on an agent (② in Fig. 4) process to invoke sequences of fuzzed system-calls. NG-Agent is the component responsible for converting binary fuzz-inputs, received from QEMU-Fuzz into data passed through system-calls, including system-call IDs, system-call arguments, and user-space memory-contents accessed by the kernel. At startup, the agent reads a config that identifies the component that should be fuzzed, and establishes communication with QEMU-Fuzz. As QEMU-Fuzz generates raw-binary inputs, NG-Agent features an interpreter that converts the raw bytes into a sequence of system-calls. The agent also launches the $\boxed{\texttt{CFU}}$ and $\boxed{\texttt{UFFD}}$ threads that are responsible for handling user-space memory access hooks. As the agent consumes bytes from the input, it assembles a "canonical" version of the input (explained in Section IV-C3c). After the input has been executed, the agent requests a VM reset and a new input from the fuzzing engine.

1) Agent Initialization: At startup, NG-Agent initializes coverage collection, inflates the process' memory, and sets up the user-memory and snapshot interfaces.

a) Configuring kcov: NG-Agent configures kcov to collect coverage over the kernel [26]. kcov is a Linux kernel code-coverage mechanism, tailored to and used by fuzzers such as Syzkaller. By enabling kcov, the NG-Agent process can access a kcov memory region which contains kernel coverage-data. kcov has two modes: it can be configured to report either the covered kernel program-counters(PCs), or covered comparison instructions along with the corresponding operands(CMP tracing). These modes are mutually exclusive (each fuzzer process can either trace PCs or comparison instructions). The information from the CMP tracing mode is useful for helping the fuzzer automatically solve constraints in the code (such as `ioctl` request number checks). As such, existing fuzzers such as Syzkaller use both modes during fuzzing. NG-Agent implements support for both coverage modes. Furthermore, FUZZNG extends kcov's CMP mode to report comparisons performed in the kernel using APIs such as `strncmp` and `memcmp`. This change is inspired by modern user-space fuzzers that can automatically identify strings that the target expects to be in the input, by e.g., hooking functions related to string and memory comparisons. In Section V-B, we show that this feature proves useful for populating fuzzing inputs with strings that the kernel expects to receive through system-call arguments. Prior to fuzzing, each NG-Agent instance (running in its own VM) queries QEMU-Fuzz to determine whether to use the PC or CMP mode.

b) Inflating Process Memory: Hooking user-space memory accesses allows FUZZNG to fuzz complex interfaces without knowledge about pointer arguments and struct layouts encoded in grammars (see Section II-A). However, normal processes in Linux map a tiny fraction of the available virtual memory space (128 TB on x86-64). As such, by default, a random fuzzer-provided address in user-memory is unlikely to be backed by any physical-memory mapping, and FUZZNG would not be able to populate those virtual-addresses with fuzzer-provided data.

As a solution, NG-Agent "inflates" its address-space by mapping as much memory as possible using the `mmap` system-call. NG-Agent leaves a small fraction (16MB) of memory unallocated so that it is possible for fuzzer-generated `mmap` system-calls to succeed. Note that though the underlying hardware cannot back the $\approx$ 128 TB of mapped virtual addresses with actual physical memory, physical-memory is only allocated when a virtual-page is first accessed. Throughout the lifetime of individual fuzzing runs, only a tiny fraction of the "inflated" memory is ever accessed and backed by physical pages. After this inflation step, virtually all valid user addresses ($< 0x800000000000$) are associated with a mapping. As such, randomly generated addresses have a high chance of hitting valid user-space memory mapped by the NG-Agent, and FUZZNG's hooks can populate memory accessed by the kernel, in response to an issued system call.

c) User-Memory Hook Service Threads: As mentioned in Section III-B, FUZZNG hooks kernel-memory accesses to userspace. To do this FUZZNG relies on two mechanisms - hooking of `copy_from_user`-type APIs, and userfaultfd. For each of these mechanisms, NG-Agent initializes threads ($\boxed{\texttt{CFU}}$ and $\boxed{\texttt{UFFD}}$ in Figure 4) which is responsible for filling memory referenced by hooked user-memory reads with bytes taken from the fuzzing input. We describe the threads' functionality in further detail in Section IV-C2.

d) Communicating with QEMU-Fuzz: NG-Agent communicates directly with QEMU-Fuzz by raising its privilege level using Linux' `iopl` and executing Port-IO instructions. Each Port-IO instruction causes an exit into QEMU-Fuzz, which handles the request. When NG-Agent has performed coverage and memory-related initialization, it is ready to execute fuzzer inputs. To start the fuzzing process, NG-Agent uses Port-IO to provide QEMU-Fuzz with the addresses of memory allocated for the fuzzer-input and for kcov coverage. As QEMU-Fuzz interacts with the VM's memory using physical addressing, the agent process uses the `/proc/self/pagemap` interface to convert virtual addresses to physical addresses. Then NG-Agent asks QEMU-Fuzz to create a VM snapshot and provide a new fuzzer-input. Once NG-Agent has interpreted the input, it requests a VM reset, and the fuzzing process repeats.

2) Agent Config: NG-Agent interprets inputs generated and provided by QEMU-Fuzz as sequences of system-calls. FUZZNG relies on an agent-config to target specific components of the kernel. We refer to logical groupings of kernel-features as "components". Examples include device drivers (console/ptmx, rdma, vhost), interfaces (KVM, io_uring), and generic APIs (bpf). The config consists of two sections:

1) *Files:* Here, we specify paths to component-specific files (in `/dev/`) that the agent opens (if any) prior to fuzzing.
2) *System-Calls:* The list of system-calls that the fuzzer can generate. For each system-call, we specify the number of arguments, and an optional "mask" for each argument.

For example, the KVM config in Figure 1, the first line specifies that the `/dev/kvm` file should be opened, as it is the entry-point for all interactions with the KVM subsystem [55]. The remaining six lines simply list the system-calls that can interact with KVM. Each line represents a system-call that the fuzzer can invoke along with the number of arguments expected by that system-call. Optionally, we can provide a mask for some arguments. The masks are not strictly necessary, but help limit slow system-calls. For example, in Figure 1, we specify masks for the read/write system-calls to restrict the maximum size of the operation. Additionally we apply masks to the mmap system-call to avoid potentially destructive mappings that overwrite coverage/code regions of the agent.

The list of system-call IDs needed to interface with a kernel-component can be easily collected by reading the Kernel documentation, or by examining the file APIs supported by an interface (e.g., examining the fields in `file_operations` structs). Compared with Syzkaller, there are practically no constraints on the system-call arguments (beyond the mask to reduce patently invalid/wasteful arguments). Instead, we rely on the input-space reshaping hooks and coverage-feedback to identify inputs with valid/interesting arguments. Each argument (including file descriptor numbers and pointer addresses) is simply taken directly from the binary fuzzer input.

3) The NG-Agent Interpreter: The input QEMU-Fuzz provides to NG-Agent is simply a byte-buffer. To convert these bytes into a sequence of system-calls, FUZZNG implements an interpreter. The interpreter uses a 4-byte value (ASCII "`FUZZ`") to split the input into individual operations. Modern fuzzers, such as *libFuzzer*, can automatically identify such "magic" values and insert them into inputs. There are two types of operations: System-calls and User-Memory Patterns.

a) System-Calls: For each operation, NG-Agent examines the first byte and uses it to select a system-call, by indexing it into the table of available system-calls, as defined in the NG-Agent config. Once NG-Agent selects the system-call, it reads the number of arguments necessary for that system-call (as specified in the agent config), applying an argument mask to each one, where specified. Finally NG-Agent, uses the `syscall()` *libc* API to invoke the system-call.

Internally, NG-Agent also adds the `fuzz-set-fd-offset` system-call to the table, so the fuzzer can specify which fd should be used to respond to `fdget` calls for unallocated fd numbers. The fuzzer often generates inputs that interact with multiple file-descriptors. For example, for an input to run a KVM virtual-machine, it must execute the `KVM_CREATE_VM` `ioctl` for the `/dev/kvm` file, which creates an fd for the VM. Then, the input must run the `KVM_CREATE_VCPU` `ioctl` for the newly created VM fd, which creates a VCPU fd. Finally, the input must execute the `KVM_RUN` `ioctl` for the VCPU fd. As mod-NG stores newly created fds on a stack, system-calls that interact with file-descriptors (such as `ioctl`), will target the last-created file-descriptor. However, if after the `KVM_RUN` call, the input tries to execute the `KVM_SET_MEMORY_REGION` `ioctl` (which is specific to VM fds), the system-call will fail, as the VCPU fd is at the top of mod-NG's stack. As such, we provide the input with the capability to call the `fuzz-set-fd-offset` system-call, in order to choose which file-descriptor on the stack to use. As the VM fd is at the 2nd position of the stack (index 1), the fuzzer call `fuzz-set-fd-offset(1)`, prior to invoking the

```
ioctl(kvm_fd, KVM_CREATE_VM(i.e. 0xae01), 0)
01 [00  0d  00  00][01  ae  00  00][00  00  00  00] 46  55  5a  5a
ioctl(vm_fd, KVM_CREATE_VCPU(i.e. 0xae41), 0)
01 [e9  0c  00  00][41  ae  00  00][00  00  00  00] 46  55  5a  5a
ioctl(vcpu_fd, KVM_SET_MSRS (i.e. 0x4008ae89),
     (struct *kvm_msrs)(0xf75afd26) = {0x1a, 0, ..})
01 [e9  09  00  00][89  ae  08  40][26  fd  5a  f7] 46  55  5a  5a
1a  00  00  00  00  00  00  00  03  4d  56  4b  46  00  00  5a ...
```

Fig. 5: A partial input found by FUZZNG while fuzzing KVM. The gray, italicized lines are comments that represent the system-call represented in the subsequent line of bytes. In the input, the red bytes (ASCII for "FUZZ"), are used to separate operations. The leftmost, blue bytes represent the system-call index. The square brackets delimit the arguments passed to the system-call. Yellow bytes are interpreted as file-descriptor numbers. Turquoise bytes represent "magic" `ioctl` request numbers. The orange bytes represent addresses that are accessed by the kernel. The purple bytes at the bottom are used to fill the kernel-access. Note that we manually added these annotations to the figure, purely for explanation purposes. FUZZNG has no inherent knowledge about argument types, etc.

`KVM_SET_MEMORY_REGION` `ioctl`. As it can take time for the fuzzer to "guess" fd offsets, for half of the fuzzer VMs in our evaluation (see § V-A) we configure the fuzzer to automatically call `fuzz-set-fd-offset` when a system-call fails (returning -1), cycling through all open fds. In this mode, the fuzzer does not need to guess fd offsets on the stack, as the agent interpreter automatically tries all possible options. This maximizes the chance that the system-call will be performed with a correct type of file-descriptor.

b) User-Memory-Accesses: Unlike system-call operations, user-memory-accesses are initiated by the kernel, rather than NG-Agent. However, in the input they are still represented as operations. When a user-memory-access occurs, it is handled either by the $\boxed{\texttt{CFU}}$ thread, or the $\boxed{\texttt{UFFD}}$ thread (depending on whether the copy_from_user-APIs were used). To fill the corresponding location in memory with fuzzed data, these threads interpret the next operation in the fuzzer-input as a user-memory pattern. Depending on the size of the access, and consequently the amount of fuzzer-provided data needed to fill the access, NG-Agent uses different strategies. For smaller accesses (fewer than 256 bytes), NG-Agent simply reads the exact number of bytes needed from the fuzzer-input, and uses them to fill the region accessed by the kernel. This amount of data is sufficient for most structs passed into the kernel. For larger accesses, NG-Agent reads the first byte of the operation and interprets it as the length of a repeating-pattern (taken from the subsequent bytes) used to fill the kernel-read. Once the $\boxed{\texttt{CFU}}$/$\boxed{\texttt{UFFD}}$ threads populate a memory region with bytes from the input, they update a global input pointer, so that the system-call interpreter running in the main thread knows to advance past the user-memory-access operations.

c) Enforcing Input Structure: Unlike approaches such as Syzkaller, whose inputs contain detailed information about system-call types and struct fields that must be populated, FUZZNG has no semantic information about fuzzing inputs that allows it to predict the system-call sequence and user-space accesses that correspond to an input. FUZZNG inputs are simply bytes separated by "FUZZ". However, as FUZZNG executes an input, it dynamically gains valuable information about the system-calls that it represents. For example, as FUZZNG interprets the input, it learns about:

- Parts of the input that are ignored/unused. E.g. extraneous system-call arguments, or masked-out parts of arguments.
- Parts of the input that are normalized to a different value. E.g. the first byte of a system-call operation is normalized relative to the size of the system-call table.
- The number of bytes needed to fill user-memory-access operations.

To leverage this valuable information, we made modifications to *libFuzzer* to support modifying inputs during fuzzing, so that inputs can be canonicalized during execution. All of the interpreter operations we described have specific length requirements. However, these requirements are often not satisfied by inputs provided by *libFuzzer*: operations might feature too many, or too few bytes. To address this, NG-Agent dynamically "resizes" operations, as it is interpreting the fuzzer input, so that each operation contains precisely the number of bytes required. Thus, NG-Agent enforces input structure, and ensures that inputs saved by *libFuzzer* do not contain wasted bytes (see Fig 5 for an example). As all bytes in stored inputs are actually used for operations, input-mutations are more likely to achieve new code coverage.

For system-calls, NG-Agent will delete operations that do not have enough bytes for all of the system-call arguments from the fuzzer-input. Conversely, if there is a surplus of bytes after an operation, FUZZNG removes the excess from the input. Similarly, FUZZNG ensures that the number of bytes in a user-memory-access operation exactly matches the amount needed to fill the access. If there are not enough bytes, FUZZNG uses a prng (seeded from rdtsc) to insert random data into the input until the operation's size matches the access. Thus the result input will contain a perfectly sized user-memory-access operation (filled with random data, if necessary), which can be mutated for future executions. Additionally, NG-Agent applies system-call argument masks directly to the input. For example, if the agent-config specifies that an argument has a mask, `0xF000`, and the fuzzer input provides `0xDEADBEEF` as the argument, NG-Agent replaces `0xDEADBEEF` with `0x0000B000` in the input. NG-Agent also normalizes (confines to a range of values) the byte used to select the type of system-call by the number of fuzzer-accessible system-calls.

After NG-Agent has executed an input, it returns the canonicalized version to QEMU-Fuzz. By default *libFuzzer* does not support modifying the length or contents of fuzzer-provided data, so we made slight modifications to allow this. The result is that inputs stored in the corpus do not contain any wasted bytes, and operation sizes are guaranteed to match the number of bytes necessary. Note that since FUZZNG stores "canonicalized" inputs, the use of a prng to resize operations does not raise issues with non-determinism: any randomly-generated bytes are stored within the inputs in the corpus.

In summary, our implementation of FUZZNG combines a snapshot-fuzzing engine used to generate inputs and reset state (QEMU-Fuzz), a user-space agent used to invoke system-calls and populate relevant memory with fuzzed data (NG-Agent), and a kernel-module(mod-NG) used to hook kernel APIs related to system-calls and provide dynamic feedback about the API invocations to the agent.

V. EVALUATION

We evaluate FUZZNG's fuzzing capabilities to answer the following research questions.

RQ1 Does FUZZNG achieve competitive coverage when compared with state of the art grammar-based system-call fuzzers? (see § V-B)
RQ2 What is the average size of FUZZNG configs, compared to Syzkaller syzlang descriptions? (see Table I)
RQ3 Can FUZZNG discover new bugs in the Linux kernel? (see § V-D)
RQ4 How does FUZZNG's snapshot-fuzzing performance compare with Syzkaller's fork-server? (see § V-E)

A. Experimental Setup

We performed all experiments on servers with Dual Socket Intel Xeon E5-2600 v3 Series CPUs, ranging between 192 and 256 GB RAM. All of our fuzzing experiments were performed against Linux Kernel 5.12. FUZZNG uses multiple QEMU-Fuzz VMs to fuzz kernel-targets on multicore systems. FUZZNG configures each VM with different fuzzing options:

1) *Coverage Mode:* As mentioned in Section IV-C1a, KCOV supports two coverage modes: PC coverage and CMP coverage. PC coverage enables FUZZNG to consistently determine when new code was reached. CMP coverage allows FUZZNG to solve input constraints, such as "magic" `ioctl` request numbers. As such, FUZZNG configures half of the fuzzer VMs with PC coverage, and the other half with CMP coverage. This achieves a balance between fuzzer instances that simply store inputs that reached new edges (PC coverage) and ones that are useful for overcoming complex value comparisons (CMP coverage).
2) *Errorless Inputs:* FUZZNG configures an eighth of all VMs to discard inputs that result in system-calls that fail (i.e. return -1). This incentivizes the fuzzer to find "high-quality" system-call sequences that reach deep kernel states, without any failed/wasted syscalls. As error code-paths should certainly be targeted, this feature is only enabled for every eighth VM.
3) *File-Descriptor "Cascading":* As described in IV-C3a, in this mode, when a system-call, involving an fd, fails (returning -1) FUZZNG uses the `fuzz-set-fd-offset` functionality to iterate through all available FDs, repeating the system-call for each one, until the system-call succeeds, or all open FDs have been tried. This reduces the need for the fuzzer to correctly guess which FD should be used by each system-call. However as many system-calls will return errors, this slows down input execution, so we only configure this mode for half of the VMs.

Note that the options are not mutually exclusive (e.g., a VM can fuzz with CMP Coverage and Cascading). All of the parallel fuzzers store new interesting inputs in the same corpus directory. As such, even if an input finds new coverage only in a single mode, it is mutated by all of the fuzzer instances.

B. Coverage

FUZZNG's main contribution is its ability to fuzz complex kernel interfaces, with minimal setup-cost for each new interface, when compared to current system-call fuzzers. Thus,

		Syzkaller		Healer	FuzzNG	
Component	Max Cov	Edge Count	Syzlang LoC	Edge Count	Edge Count	Config LoC
bpf	15359	3623	864	1121	3572	1
video4linux	1004	563	381	446	567	4
rdma	4014	562	1474	*	591	5
binder	2506	340	272	56	344	6
cdrom	956	138	351	120	144	5
kvm	34924	9213	891	8755	9468	7
vhost_net	415	218	157	210	225	9
drm	12503	2296	745	1978	2138	7
io_uring	3413	982	343	986	1003	6
vt_ioctl	332	142	381	138	162	9
Average				76.52%	**102.53%**	**1.67%**
Geo. Mean				66.95%	**102.41%**	**1.09%**
vs. Syzkaller						

TABLE I: Coverage comparison of FuzzNG, Syzkaller and Healer. Note that, for consistency, the "Syzlang" LoC column excludes any additional harnesses (written in C and Go), which Syzkaller relies on. Coverage is averaged over 5 runs

we compared FuzzNG's coverage performance against the de-facto Linux kernel fuzzer, Syzkaller, and Healer which applies relation-learning techniques to Syzkaller's grammars to improve mutation efficiency.

We configured all three fuzzers to fuzz the same kernel builds. For each Kernel component fuzzed, we provided a separate coverage allowlist, which only applies the KCOV coverage instrumentation to the source files used to implement the component (e.g., KVM, bpf, etc.). As such, FuzzNG, Syzkaller, and Healer only reported coverage (and stored inputs) for system-calls that interacted with the relevant component under test. We select components for fuzzing according to the following criteria:

- Syzkaller (and, by proxy, Healer) must support the component (i.e., the Syzkaller repo contains relevant manually-written descriptions).
- The component must be available on x86-64 Linux.
- The component must be a system-call interface[2].
- The descriptors for the component must be comprehensive. To ensure fairness and generality of the comparison, we sorted the components by syzlang description size, and selected components based on the largest descriptor-files that fuzz a clearly-defined interface.

We fuzzed every component on 20 cores, for 168 hours (7 days). Our edge coverage results (averaged over 3 runs) are presented in Table I. FuzzNG achieved more coverage than Syzkaller for seven components. For the remaining three components, FuzzNG's edge coverage was within 7% of Syzkaller's. On average, FuzzNG achieves 102.5% of Syzkaller's coverage.

Healer did not achieve any coverage over the RDMA code as it relies on an older version of Syzkaller that does not contain RDMA descriptions. We found that the open-source version of Healer (2efbb44c7d) achieves a lower coverage than Syzkaller, for the components we evaluated. Healer is designed to efficiently identify relations between system-calls. In our kernel-configurations, only the target component is instrumented for coverage. Thus only system-calls that increase coverage over the target component are added to the corpus. As such, Healer's improvement to Syzkaller is limitted since the fuzzer naturally targets a specific component. Additionally, the repo maintainers mention that the open-source version of Healer has many limitations compared with the private version of Healer used in the Healer paper, which likely accounts for the coverage difference [52], [3].

We compared the coverage/edges only reached by Syzkaller and Healer against FuzzNG. The results for each component are presented in Figure 6. We manually inspected the coverage and found several common causes for edges covered by Syzkaller/Healer that FuzzNG did not cover. Syzkaller and Healer support *fault-injection*, allowing the fuzzer to force failures in kernel-API calls (e.g., SLAB allocation, futex). We did not implement this feature for FuzzNG. As such, some error handling code in the kernel is not covered by FuzzNG, but is covered by Syzkaller. Furthermore, Syzkaller (and Healer which uses Syzkaller's executor) can use *multiple threads* to execute a testcase. Currently, FuzzNG runs all system-calls from a single-thread. As such, parts of components that are responsible for task reference-counting are only covered by Syzkaller. Much of the code in KVM that only Syzkaller covered is related to instruction emulation of VMs in different x86 operating modes (real-mode, protected-mode, and long-mode). Instructions are emulated differently by KVM, depending on the CPU mode. However, as the setup for a real-mode VM is considerably different from a long-mode VM, there is no feedback to guide input-generation towards different emulation contexts. Syzkaller uses virtual system-calls that explicitly encode the operating mode, which do not require complex mutations to reach the code. However, FuzzNG is able to fully-cover some instruction emulation routines (in all x86 modes), so it is possible that FuzzNG would cover more of this code, given additional time. In BPF, we found that FuzzNG's libfuzzer-based mutators was slow to generate valid BPF programs. FuzzNG quickly found a way to generate the smallest possible valid 16-byte BPF program, however as generating a longer program requires simultaneously inserting bytes into the BPF program, and increasing the length-field describing the program size, FuzzNG was unable to generate large BPF programs, hampering coverage. In the future this can be addressed by making the mutations aware of length-fields, which FuzzNG can identify by correlating byte-values with user-space-access lengths.

However, as shown in Fig. 6, FuzzNG also covers parts of the code, which Syzkaller+Healer do not for all components but binder. Additionally, ensemble-fuzzing has been shown to outperform individual fuzzers [11]. As such, it will be beneficial to fuzz using an ensemble of FuzzNG and Syzkaller-based techniques. Since FuzzNG and Syzkaller both have their own representations of inputs, collaborative fuzzing will require an adapter that can translate between the input formats.

Additionally, we monitored coverage achieved, as time progressed while fuzzing KVM for both Syzkaller and FuzzNG. The results are presented in Figure 7. As expected Syzkaller initially significantly outperforms FuzzNG, due to its comprehensive grammar suite, however the coverage quickly plateaus. Noticeably, FuzzNG's grammarless approach eventually overtakes Syzkaller, by the 60th hour of fuzzing. The potential benefits from running FuzzNG are obvious, as coverage-gains are not limited by manually-written grammars. FuzzNG's initial "lag" in coverage is expected. However, as FuzzNG stores corpus-inputs, subsequent fuzzing runs will start at high coverage, making the lag a nonissue after the initial run. On the contrary, FuzzNG's lack of dependence on grammars allows it

[2]E.g., Syzkaller supports fuzzing incoming Bluetooth packets. As these packets originate from devices, rather than system-calls, we consider bluetooth fuzzing outside of the scope for this paper.

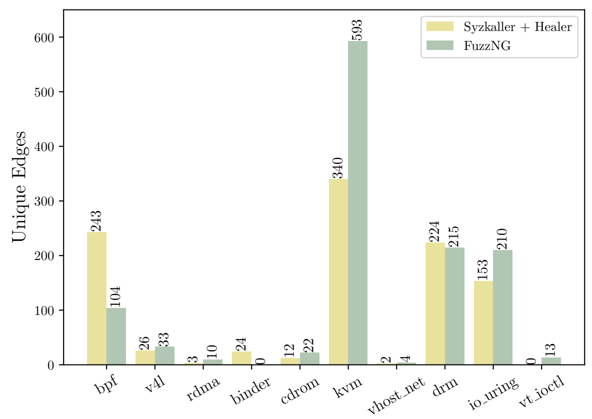

Fig. 6: Edges uniquely covered by Syzkaller+Healer and FuzzNG

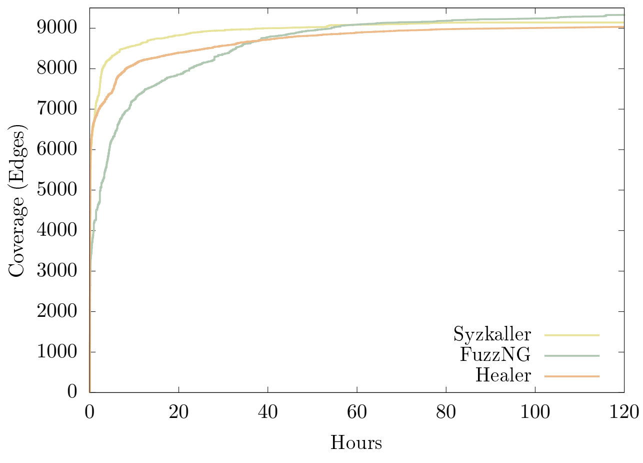

Fig. 7: Coverage comparison between Syzkaller and FuzzNG, while fuzzing KVM, over time. Syzkaller rapidly achieves high coverage over KVM, due to its extensive descriptions. However, eventually FuzzNG overtakes Syzkaller.

to continue discovering new code, without human intervention.

C. Config-Size

We compare the size of FuzzNG's configs and Syzkaller's syzlang descriptions in Table I. This comparison represents the implementation cost of adding support for new components to each fuzzer. Note that for Syzkaller we only include the syzlang descriptions, and omit any component-specific harnesses, as this code is interleaved with unrelated Syzkaller code. As such, we underestimate the amount of lines needed to add support for new components to Syzkaller. On average, FuzzNG's configs are 98.3% smaller than Syzkaller's descriptions.

D. Bug-finding

FuzzNG's coverage evaluation primarily focuses on components that have extensive Syzkaller descriptions, and have been continuously fuzzed for years with Syzkaller. As such, we do not expect to find many bugs in this code. Nonetheless, FuzzNG found previously-unknown bugs in code fuzzed by Syzkaller. Furthermore, we picked 3 drivers that are not fuzzed by Syzkaller (mmcblk, megaraid, nvme) and created FuzzNG configurations for them (17 lines of configuration, in total). In total, FuzzNG found 9 previously-unknown bugs in the Linux Kernel (see Appendix). Of these, 5 were in components that had syzlang descriptions, and were well-covered by Syzkaller. FuzzNG found bugs in all three components that do not have Syzkaller descriptions. All bugs were found during the 120 hour measurement campaigns and are undergoing responsible disclosure. Next we discuss three case studies.

Null-ptr dereference in KVM Emulation Code: FuzzNG found a null-pointer dereference bug in KVM's `emulate_int` code, responsible for emulating interrupts. To find this bug, FuzzNG had to use independent ioctl calls to create a VM, create a VCPU, create a memory slot for the VM and launch the VM. When the CPU accessed the VM's memory to run CPU instructions, FuzzNG populated the corresponding region with fuzzer data. The fuzzer-generated instructions that trapped and caused a VMEXIT to KVM's emulation code, where a null-pointer dereference occurs due to a malformed virtualization context. Although Syzkaller has full coverage over the `emulate_int` code, Syzkaller did not find this bug, as the VM setup is handled entirely by a hard-coded harness, which sets up registers and page-tables, and the instructions executed within the VM are generated by an instruction generator designed to create well-formed sequences of instructions. FuzzNG does not rely on any KVM-specific harnessing. As such, though it takes FuzzNG longer to achieve the same coverage as Syzkaller, FuzzNG's mutator has full control over the invoked system-calls, rather than being fundamentally limited by descriptions and harnesses. This allows FuzzNG to find bugs in code that other fuzzers could not thoroughly exercise, due to rigid grammars.

Null-ptr dereference in io_uring task exit code: FuzzNG found a null-pointer dereference bug in io_uring thread-manager code. If the process, which invokes io_uring, raises an abort signal while the io_uring thread-manager code is executing, there is a potential race condition exposed, as the thread-manager code updates the task's IO-related bitmap (triggering a null-pointer dereference). NG-Agent aborted the process, while in the userfaultfd thread, triggering the bug. Though Syzkaller fuzzes io_uring, it did not catch the bug.

Use-after-free in megaraid code: FuzzNG found a use-after-free bug in driver code for the MegaRAID SAS RAID controllers. The MegaRAID driver uses `instance` structs to track state for individual MegaRAID devices. FuzzNG created an input that generated an ioctl that caused an `instance` to be freed, prior to invoking a management command that attempted to write a DMA-related physical-address into the freed struct, triggering the error. Notably, Syzkaller does not feature any descriptions for this device, and consequently, has not found this bug.

E. Executions Per Second

FuzzNG implements VM-snapshot fuzzing, to facilitate cleanup between test-cases. Unlike FuzzNG, Syzkaller uses a light-weight fork-server approach. We fuzzed bpf, a complex, but mostly hardware-independent interface, with both FuzzNG and Syzkaller for 24 hours on 4 cores. We found that on average FuzzNG executes 154 test-cases per-second, per core, while, Syzkaller executes 177 test-cases per-second, per core. Inspecting total executions across all fuzzed components, we found no significant deviations from this result. As such FUZZNG's comprehensive, VM-snapshotting approach achieves comparable performance to Syzkaller. Furthermore, this demonstrates that FUZZNG's coverage, when compared with Syzkaller, stems from its reshaping of the input space, rather than some vast differences in execution rate.

VI. DISCUSSION

Despite FUZZNG's positive results, we briefly discuss limitations and avenues of further improvement.

A. Other Kernel Fuzzers

Kernels have been a major target for fuzzing. Most kernel fuzzers rely on fine-grained manually-written or inferred system-call grammar. FUZZNG proposes a runtime-hooking technique to avoid the need for detailed grammars. However, other fuzzers have examined different parts of the kernel-fuzzing problem space. Here we discuss whether FUZZNG is compatible with existing fuzzing approaches.

Moonshine collects and distills system-call traces (from `strace`) and converts them into seeds that can be used with Syzkaller. Future works can apply Moonshine to generate seeds for FUZZNG (the Moonshine paper states that adding support for non-Syzkaller fuzzers is straightforward) [38].

Healer uses relation-learning to improve the efficiency of Syzkaller's mutations [38]. Our experiments in Section V-B showed that Healer's benefit is limited when fuzzing individual components. However, if future-works extend FUZZNG to fuzz the entire kernel, Healer's techniques could be used to learn the relationships between system-calls generated and executed by FUZZNG, improving fuzzing efficiency.

Other kernel fuzzers, such as *Difuze* and *SyzGen* aim to automatically recover Syzlang descriptions for interfaces. These approaches rely on separate stages to generate interface grammars and fuzz using the grammars. Instead FUZZNG operates in a single stage, reshaping the input-space to fuzz in a single step - eliding the need for detailed grammars.

Difuze performs static-analysis over kernel code in order to automatically infer descriptions for ioctl-based device-interfaces [12]. In particular, Difuze places a special emphasis on recovering, ioctl command values and argument types (by performing inter-procedural type-propogation for operands to `copy_from_user` arguments). FUZZNG performs runtime hooking of `copy_from_user` operations and automatically infers ioctl command values by collecting KCOV CMP coverage. However future work could extend FUZZNG with a static-stage inspired by Difuze that autoamtically generates FUZZNG-config variations (e.g., by identifying all of the system-calls that can interact with a subsystem).

SyzGen targets closed-source kernels (MacOS) and applies symbolic execution to automatically recover grammars for interfaces. SyzGen recovers argument types such as strings, byte-arrays, pointers, and length fields. SyzGen outputs Syzkaller descriptions for recovered interfaces. Since FUZZNG reshapes the system-call input-space, it can fuzz pointers and arrays, transparently, with runtime-hooking. SyzGen's symbolic-execution techniques can also automatically infer integer-argument ranges, and integer arguments that represent flag-fields, however we found that off-the-shelf fuzzing engines, such as libfuzzer, perform well without annotations for these fields. Nonetheless, SyzGen's techniques could be used to automatically provide feedback about types of arguments to FUZZNG (without creating explit grammars), which could potentially boost fuzzing efficiency.

B. Full-Kernel Fuzzing

A default deployment of Syzkaller fuzzes all system-calls that have syzlang descriptions. This approach is able to reach lines (and find bugs) that require interacting with multiple kernel-components, simultaneously. Currently, FUZZNG cannot arbitrarily open named files, and we rely on configs and coverage-filters to focus the fuzzer on individual components. However, recent works show that it is possible to apply ”relation learning” to automatically infer supported system-calls associated with each file and common sequences of system-calls, by observing coverage [52]. By applying the same techniques to FUZZNG, and tuning the mutator to detect and generate meaningful input-sequences it may be possible to fuzz the kernel without limiting the fuzzer to individual components, or relying on configs.

VII. RELATED WORK

Fuzzing has gained widespread attention in the academic community. In this section, we provide a brief overview of work related to kernel-fuzzing. A major catalyst reviving interest in fuzzing, was the release of the American Fuzzy Lop (AFL) [62] fuzzer, which popularized, coverage-guided, fuzzing for a wide range of software. Researchers have focused on improving fuzzing performance, with advancements in input scheduling [25], [58], [43], mutation algorithms [35], [8], [42], and input feedback [4], [63], [17]. Other systems focus on applying concolic execution [61], [29], [28] to overcome roadblocks, such as comparisons against “magic constants”, and checksums [41]. Fuzzers such as AFL with laf-intel[30] and libFuzzer[48] have applied source-code instrumentation to identify comparisons against magic bytes and produce inputs that can pass them. Other works have adapted fuzzers to complex targets such as code-interpreters [60], [56], [22], [19], compilers [31], [10], [34], network-protocols [6], [16], [13], and virtual-devices [20], [37], [46], [45], [7]. V-Shuttle[39] has demonstrated that complex hypervisors can be fuzzed without grammars, by hooking key direct-memory-access APIs.

Recently, snapshot-based fuzzing has gained traction, especially for large, stateful, fuzzing-targets. Agamotto introduces high-performance snapshots for fuzzing, based on QEMU [50]. Agamotto supports creating multiple snapshots at different points within the target's execution, to accelerate fuzzing. Nyx builds upon QEMU/KVM to implement rapid register, memory and virtual-device snapshots for fuzzing [45]. Similarly, FUZZNG implements a simple QEMU-based snapshot-fuzzer, QEMU-Fuzz, tailored towards fuzzing the Linux kernel and accepting KCOV-formatted coverage information.

Operating system kernels have received widespread attention within the academic community, with fuzzing systems purpose-built for kernel race-conditions [24], file-systems [59], and peripheral interfaces [49]. Similarly, VIA [21] fuzzes OS-drivers to identify bugs that could compromise security-guarantees in a confidential-computing environment, where virtual-device code is not trusted. kAFL introduces hardware-based coverage-collection mechanisms in order to perform coverage-guided fuzzing of OS kernels, without source instrumentation [47]. Unlike these works, FUZZNG focuses on reducing the need for descriptions and harnesses for generic system-call fuzzing.

The system-call interface has received the most attention within the OS fuzzing community. Starting in the 90s, there have been multiple fuzzers created that operate simply by generating random-arguments, such as tsys, iknowthis, sysfuzz, xnufuzz, and kg_crashme [54]. System-call fuzzers such as Trinity improved the naive system-call generation algorithms by incorporating system-call descriptions[54]. As coverage-guided fuzzers gained traction for userspace applications, Syzkaller was created to combine the strengths of description-based fuzzers with coverage-guidance for fuzzing Linux. Today, Syzkaller is the most popular system-call fuzzer, has been incorporated into the Linux Kernel development cycle and has been ported to OSes, such as XNU, FreeBSD, and Windows [14]. Syzkaller has reported thousands of bugs to the Linux Kernel developers and has become a crucial part of the kernel development lifecycle. Unlike past approaches, FUZZNG leverages kernel-hooks to achieve coverage that is comparable to Syzkaller without the need for detailed system-call descriptions.

Several works aim to automatically generate system-call descriptions. Difuze performs static-analysis over kernel code in order to automatically infer descriptions for device-interfaces for fuzzing [12]. IMF relies on kernel API-interaction logs collected with the help of application hooks, to infer grammars for macOS system-calls[18]. SyzGen relies on data-mining and symbolic-execution of manually-collected log-traces to automatically create macOS system-call grammars [9]. Notably, all of these systems focus on automatic system-call description generation, however none of these works performed comparisons with Syzkaller for interfaces with well-defined, manual specifications. KSG uses symbolic-execution to automatically generate syzlang descriptions that achieve competitive coverage compared with Syzkaller, however the source-code has not been released [51]. Unlike grammar-generation techniques, FUZZNG reshapes the kernel's input-space to make it conducive to fuzzing, rather than requiring seed-traces and relying on extensive static/dynamic analysis stages.

Other academic works have focused on improving Syzkaller's performance, without directly generating grammars. Moonshine relies on seed traces of system-calls from real-world programs to improve Syzkaller's descriptions [38]. Healer applies relation-learning to improve Syzkaller's system-call sequence mutation algorithm. SyzVegas leverages machine learning techniques to improve Syzkaller's coverage [57]. Agamotto leverages dynamic VM snapshots to skip executing system-calls that are common among Syzkaller inputs, increasing fuzzing throughput. HFL extends Syzkaller with symbolic execution [27]. Unlike these approaches, FUZZNG does not depend on manual or "learned" descriptions of system-call behaviors. Instead, FUZZNG combines the techniques from the older random-argument fuzzers with new coverage-guided techniques and reshapes the system-call input-space to create a fuzzing system that achieves competitive coverage, when compared with description-based approaches.

VIII. CONCLUSION

FUZZNG is the first fuzzer capable of producing complex system-call interactions without manually written system-call descriptions or prior analysis over source-code/seed programs. FUZZNG relies on fundamental properties of OS kernels in order to "reshape" the system-call interface, removing the fuzzing-roadblocks created by pointer and file-descriptor arguments. At its core, FUZZNG simply interprets binary inputs from a general-purpose fuzzing-engine (libFuzzer) into sequences of system-calls. Hooks implemented in mod-NG transparently allow FUZZNG to populate file-descriptors and complex data-structures just in time for the kernel to access them. The evaluation of our FUZZNG prototype shows that it achieves 102.5% of Syzkaller's coverage, with only 1.7% as many lines of per-component configuration code. Furthermore, as FUZZNG's does not rely on any component-specific harnesses, which can make bugs unreachable, FUZZNG found bugs in functions that were covered by Syzkaller. Additionally, even though our evaluation focused on well-tested components of the Linux Kernel that have been fuzzed with high-coverage, for years, we found 9 previously unknown bugs that we are responsibly disclosing. We will open-source all FUZZNG code and work with upstream efforts to integrate FUZZNG so that it can continue benefiting the Linux Kernel community.

ACKNOWLEDGEMENTS

We would like to thank our anonymous reviewers for their insightful comments and feedback. We acknowledge that the computational work reported in this paper was performed on the Shared Computing Cluster, administered by Boston University's Research Computing Services and on CloudLab [15]. This work was supported by the National Science Foundation (NSF) under grant CNS-1942793 and through a Red Hat Collaboratory Research Incubation Award (2023-01-RH11).

REFERENCES

[1] Designing the api: Other considerations. https://www.kernel.org/doc/html/v5.0/process/adding-syscalls.html#designing-the-api-other-considerations.

[2] Designing the api: Planning for extension. https://www.kernel.org/doc/html/v5.0/process/adding-syscalls.html#designing-the-api-planning-for-extension.

[3] Reproducing evaluation part of paper healer #37. https://github.com/SunHao-0/healer/issues/37.

[4] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. IJON: Exploring Deep State Spaces via Fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2020.

[5] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[6] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr. In *Proceedings of the International Conference on Information Security (ISC)*, Samos, Greece, 2006.

[7] Alexander Bulekov, Bandan Das, Stefan Hajnoczi, and Manuel Egele. Morphuzz: Bending (input) space to fuzz virtual devices. In *Proceedings of the USENIX Security Symposium*, Boston, MA, August 2022. USENIX Association.

[8] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, 2015.

[9] Weiteng Chen, Yu Wang, Zheng Zhang, and Zhiyun Qian. Syzgen: Automated generation of syscall specification of closed-source macos drivers. In *ACM CCS*, 2021.

[10] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Seattle, WA, 2013.

[11] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. EnFuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *Proceedings of the USENIX Security Symposium*, pages 1967–1983, Santa Clara, CA, August 2019. USENIX Association.

[12] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, Dallas, TX, 2017.

[13] Joeri De Ruiter and Erik Poll. Protocol State Fuzzing of TLS Implementations. In *Proceedings of the USENIX Security Symposium*, Washington, DC, 2015.

[14] David Drysdale. Coverage-guided kernel fuzzing with syzkaller. *Linux Weekly News*, 2:33, 2016.

[15] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The design and operation of {CloudLab}. In *2019 USENIX annual technical conference (USENIX ATC 19)*, pages 1–14, 2019.

[16] Serge Gorbunov and Arnold Rosenbloom. Autofuzz: Automated network protocol fuzzing framework. *International Journal of Computer Science and Network Security*, 10(8):239, 2010.

[17] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *Proceedings of the USENIX Security Symposium*, Washington, DC, 2013.

[18] HyungSeok Han and Sang Kil Cha. Imf: Inferred model-based fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2345–2358, 2017.

[19] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *Proceedings of Network and Distributed Systems Security Symposium (NDSS)*, San Diego, CA, 2019.

[20] Andrew Henderson, Heng Yin, Guang Jin, Hao Han, and Hongmei Deng. VDF: Targeted evolutionary fuzz testing of virtual devices. In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, Atlanta, GA, 2017.

[21] Felicitas Hetzelt, Martin Radev, Robert Buhren, Mathias Morbitzer, and Jean-Pierre Seifert. Via: Analyzing device interfaces of protected virtual machines. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[22] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Proceedings of the USENIX Security Symposium*, Bellevue, WA, 2012.

[23] kasan: add hardware tag-based mode for arm64. https://lwn.net/Articles/838211/.

[24] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, 2019.

[25] Siddharth Karamcheti, Gideon Mann, and David Rosenberg. Adaptive grey-box fuzz-testing with thompson sampling. In *Proceedings of the ACM Workshop on Artificial Intelligence and Security*, Toronto, Canada, 2018.

[26] kcov: code coverage for fuzzing. https://www.kernel.org/doc/html/latest/dev-tools/kcov.html.

[27] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. Hfl: Hybrid fuzzing on the linux kernel. In *NDSS*, 2020.

[28] Su Yong Kim, Sungdeok Cha, and Doo-Hwan Bae. Automatic and lightweight grammar generation for fuzz testing. *Computers & Security*, 36:1–11, 2013.

[29] Su Yong Kim, Sangho Lee, Insu Yun, Wen Xu, Byoungyoung Lee, Youngtae Yun, and Taesoo Kim. Cab-fuzz: Practical concolic testing techniques for COTS operating systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, 2017.

[30] Circumventing fuzzing roadblocks with compiler transformations. https://lafintel.wordpress.com/, 2016.

[31] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F Donaldson. Many-core compiler fuzzing. *ACM SIGPLAN Notices*, 50(6):65–76, 2015.

[32] Linux kernel cve data analysis vulnerabilities by version. https://blog.kernelcare.com/linux-kernel-cve-data-analysis-part-3-vulnerabilities-by-version.

[33] Linux in 2020: 27.8 million lines of code in the kernel, 1.3 million in systemd. https://www.linux.com/news/linux-in-2020-27-8-million-lines-of-code-in-the-kernel-1-3-million-in-systemd/.

[34] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, Honolulu, HI, 2019.

[35] Charlie Miller, Zachary NJ Peterson, et al. Analysis of mutation and generation-based fuzzing. *Independent Security Evaluators, Tech. Rep*, 56:127–135, 2007.

[36] Gsoc reports: Enhancing syzkaller support for netbsd. https://blog.netbsd.org/tnf/entry/gsoc_reports_enhancing_syzkaller_support1.

[37] Tavis Ormandy. An empirical study into the security exposure to hosts of hostile virtualized environments. taviso.decsystem.org/virtsec.pdf, 2007.

[38] Shankara Pailoor, Andrew Aday, and Suman Jana. MoonShine: Optimizing OS fuzzer seed selection with trace distillation. In *Proceedings of the USENIX Security Symposium*, pages 729–743, Baltimore, MD, August 2018. USENIX Association.

[39] Gaoning Pan, Xingwei Lin, Xuhong Zhang, Yongkang Jia, Shouling Ji, Chunming Wu, Xinlei Ying, Jiashui Wang, and Yanjun Wu. V-shuttle: Scalable and semantics-aware hypervisor virtual device fuzzing. In *Proceedings of the Annual Computer Security Applications Conference*, 2021.

[40] Hui Peng and Mathias Payer. Usbfuzz: A framework for fuzzing {USB} drivers by device emulation. In *Proceedings of the USENIX Security Symposium*, pages 2559–2575, 2020.

[41] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: Fuzzing by Program Transformation. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[42] Mohit Rajpal, William Blum, and Rishabh Singh. Not all bytes are equal: Neural byte sieve for fuzzing. *arXiv preprint arXiv:1711.04596*, 2017.

[43] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *Proceedings of the USENIX Security Symposium*, San Diego, CA, 2014.

[44] Rust takes a major step forward as linux's second official language. https://www.zdnet.com/article/rust-takes-a-major-step-forward-as-linuxs-second-official-language.

[45] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *Proceedings of the USENIX Security Symposium*, Vancouver, BC, 2021.

[46] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. HYPER-CUBE: High-Dimensional Hypervisor Fuzzing. In *Proceedings of the Network and Distributed Security Symposium (NDSS)*, San Diego, CA, 2020.

[47] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. Kafl: Hardware-assisted feedback fuzzing for OS kernels. In *Proceedings of the USENIX Security Symposium*, Vancouver, CA, 2017.

[48] Kostya Serebryany. libFuzzer–a library for coverage-guided fuzz testing. https://releases.llvm.org/10.0.0/docs/LibFuzzer.html, 2015.

[49] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary. In *Proceedings of the Network and Distributed Security Symposium (NDSS)*, San Diego, CA, 2019.

[50] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent Byunghoon Kang, Jean-Pierre Seifert, and Michael Franz. Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints. In *Proceedings of the USENIX Security Symposium*, Boston, MA, 2020.

[51] Hao Sun, Yuheng Shen, Jianzhong Liu, Yiru Xu, and Yu Jiang. KSG: Augmenting kernel fuzzing with system call specification generation. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 351–366, Carlsbad, CA, July 2022. USENIX Association.

[52] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. Healer: Relation learning guided kernel fuzzing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 344–358, 2021.

[53] sys/linux: automatic syscall interface extraction. https://github.com/google/syzkaller/issues/590.

[54] Trinity: Linux system call fuzzer. https://github.com/kernelslacker/trinity.

[55] Using the kvm api. https://lwn.net/Articles/658511/.

[56] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, Heraklion, Greece, 2016.

[57] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V. Krishnamurthy, and Nael Abu-Ghazaleh. SyzVegas: Beating kernel fuzzing odds with reinforcement learning. In *Proceedings of the USENIX Security Symposium*, pages 2741–2758. USENIX Association, August 2021.

[58] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Berlin, Germany, 2013.

[59] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, 2019.

[60] Dingning Yang, Yuqing Zhang, and Qixu Liu. Blendfuzz: A model-based framework for fuzz testing programs with grammatical inputs. In *Proceedings of the IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, Liverpool, England, UK, 2012.

[61] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the USENIX Security Symposium*, Baltimore, MD, 2018.

[62] Michal Zalewski. American fuzzy lop. https://lcamtuf.coredump.cx/afl/, 2014.

[63] Gen Zhang, Xu Zhou, Yingqi Luo, Xugang Wu, and Erxue Min. Ptfuzz: Guided fuzzing with processor trace feedback. *IEEE Access*, 6:37302–37313, 2018.

APPENDIX

Type of bug	Affected kernel function
`Null-dereference`	`io_sq_thread`
`Null-dereference`	`scsi_queue_rq`
`Null-dereference`	`emulate_int`
`Null-dereference`	`megasas_complete_cmd`
`Use-after-free`	`fb_mode_is_equal`
`Use-after-free`	`megasas_mgmt_fw_ioctl`
`Double-fault`	`kvm_enter`
`Assertion failure`	`sdhci_adma_table_pre`
`General protection fault`	`nvme_submit_cmds`

TABLE A1: New bugs found by FUZZNG