HFL: Hybrid Fuzzing on the Linux Kernel

Kyungtae Kim[†] Dae R. Jeong[‡] Chung Hwan Kim[¶] Yeongjin Jang[§] Insik Shin[‡] Byoungyoung Lee[*†]

[†]*Purdue University* [‡]*KAIST* [¶]*NEC Laboratories America*
[§]*Oregon State University* [*]*Seoul National University*

[†]*kim1798@purdue.edu* [‡]*{dae.r.jeong, insik.shin}@kaist.ac.kr* [¶]*chungkim@nec-labs.com*
[§]*yeongjin.jang@oregonstate.edu* [*]*byoungyoung@snu.ac.kr*

***Abstract*—Hybrid fuzzing, combining symbolic execution and fuzzing, is a promising approach for vulnerability discovery because each approach can complement the other. However, we observe that applying hybrid fuzzing to kernel testing is challenging because the following unique characteristics of the kernel make a naive adoption of hybrid fuzzing inefficient: 1) having indirect control transfers determined by system call arguments, 2) controlling and matching internal system state via system calls, and 3) inferring nested argument type for invoking system calls. Failure to handling such challenges will render both fuzzing and symbolic execution inefficient, and thereby, will result in an inefficient hybrid fuzzing. Although these challenges are essential to both fuzzing and symbolic execution, to the best of our knowledge, existing kernel testing approaches either naively use each technique separately without handling such challenges or imprecisely handle a part of challenges only by static analysis.**

To this end, this paper proposes HFL, which not only combines fuzzing with symbolic execution for hybrid fuzzing but also addresses kernel-specific fuzzing challenges via three distinct features: 1) converting indirect control transfers to direct transfers, 2) inferring system call sequence to build a consistent system state, and 3) identifying nested arguments types of system calls. As a result, HFL found 24 previously unknown vulnerabilities in recent Linux kernels. Additionally, HFL achieves 15% and 26% higher code coverage than Moonshine and Syzkaller, respectively, and over kAFL/S2E/TriforceAFL, achieving even four times better coverage, using the same amount of resources (CPU, time, etc.). Regarding vulnerability discovery performance, HFL found 13 known vulnerabilities more than three times faster than Syzkaller.

I. Introduction

Fuzzing and symbolic execution are two representative program testing techniques for vulnerability discovery. Fuzzing generates a random input to test the target program (or kernel) in hopes that such a random input triggers a corner case exhibiting a vulnerability [21, 46, 52]. Often augmented with a coverage-guided feature, random fuzz testings such as [21] have been shown its effectiveness in finding vulnerabilities in a vast number of complex real-world applications. However, such random testing is limited in handling a tight branch condition because generating an input to explore such a branch requires guessing a single value out of a huge search space [30, 36, 38].

On the other hand, symbolic execution takes a deterministic and complete testing approach [9, 13], unlike fuzzing which relies on a random approach. In particular, symbolic execution takes all input to the target program as symbols and keeps tracking the program's execution context as symbolic expressions. By interpreting an execution as a set of symbolic constraints, symbolic execution can easily handle the tight branch condition mentioned above; simply solving constraints will generate an input satisfying the branch condition. Unfortunately, symbolic execution suffers from a critical limitation, and that is state explosion. In particular, when symbolic execution faces a branch, it has to explore both sides of a branch (taken and not taken), and thereby, its search complexity grows exponentially. As a result, the application scope of symbolic execution is usually limited to small size programs [11, 15, 28, 37].

Given such advantages and disadvantages of random fuzzing and symbolic execution, hybrid fuzzing [32, 34] can be a general extension of these two. Hybrid fuzzing combines fuzzing and symbolic execution so as to complement the aforementioned limitations of each approach. In other words, when random fuzzing is blocked by a tight branch condition, symbolic execution comes to the rescue; when symbolic execution suffers from the state explosion issue, random fuzzing can assist to pinpoint a specific path to explore and thus avoid state explosion. By taking both approaches' advantages, hybrid fuzzing has demonstrated its effectiveness in discovering vulnerabilities [44, 51, 53].

However, our observation is that applying hybrid fuzzing to kernel testing is challenging mainly due to the inherent characteristics of the kernel[1]. In the following, we summarize three kernel-specific challenges that make hybrid fuzzing ineffective.

First, the Linux kernel uses many indirect control transfers to support polymorphism, and this renders traditional testing inefficient. The kernel is designed to support a large number of different devices and features, and thus throughout the kernel, it is common to see polymorphism using a function pointer table. However, random fuzzing cannot efficiently determine a specific index value fetch the function pointer table if that

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.24018
www.ndss-symposium.org

[1] The discussion throughout this paper particularly focuses on the Linux kernel, but most of descriptions and knowledge can be generally applied to other kernels as well. If not mentioned specifically, the kernel implicates the Linux kernel in this paper.

index comes from an input. Additionally, symbolic execution can neither easily handle such a case because indexing a table with symbols results in symbolic dereference and it requires the exploration of the entire value space by the symbols.

Second, it is difficult to infer the right sequence (and dependency) of syscalls[2] so as to build the system states required for triggering a vulnerability. Unlike userland programs, the kernel maintains its internal system states during its lifetime and the same syscalls may perform differently depending on its invocation context. Therefore, if fuzzing invokes a syscall without setting up the right pre-context of system states (which should have been performed through syscalls as well), the syscall would be early rejected by the kernel, hindering the deep exploration of the kernel code. Symbolic execution does not handle this issue either since the kernel has many data variables to maintain such system states and they may cause state explosion issues.

Third, nested structures in syscall arguments make interface templating difficult. Certain syscalls take a large size of the input from the user space and they use a nested structure in their arguments — i.e., a field member in one structure points to another structure. From the perspective of random fuzzing, such a nested structure is difficult to construct as it has to correctly guess the internal semantics imposed in the nested structure (i.e., a pointer pointing to another or a length field indicating the size of the buffer).

These challenges make hybrid fuzzing difficult, and to the best of our knowledge, existing kernel testing approaches [26, 35, 40, 43, 49] either naively use each technique separately by not handling such challenges or imprecisely handle a part of challenges only by static analysis [16, 24, 33].

To resolve this, this paper proposes HFL, which takes a hybrid fuzzing approach to test the Linux kernel. In particular, it addresses the aforementioned kernel-specific challenges for efficient hybrid fuzzing: 1) HFL converts indirect control-flows to direct ones, through translating the original kernel at the compilation time; 2) HFL reconstructs system states by inferring the right calling sequence. Specifically, to reduce the scope of symbolic variables, HFL performs static points-to analysis beforehand such that it can selectively symbolize data variables involved in system states; and 3) HFL retrieves nested syscall arguments at runtime by exploiting the domain knowledge on how the kernel handles the arguments.

We implemented HFL based on well-known kernel fuzzer `Syzkaller` and symbolic executor `S2E`. Then we evaluated various aspects of HFL in finding vulnerabilities of the Linux kernel. First, with regards to vulnerability finding capability, HFL discovered *24 previously unknown vulnerabilities*, 17 of which are accordingly confirmed by the Linux kernel community. In order to compare HFL with state-of-the-art kernel fuzzers, we also performed a detailed evaluation as follows. In terms of code coverage, HFL performs better than Moonshine and Syzkaller, overall 15% and 26%, respectively. Compared to kAFL, TriforceAFL and S2E, we observed that the coverage improvement of HFL is more than 4 times. To compare the vulnerability finding capability, we tested how long it takes to uncover known 13 crashes in the Linux kernel. Our results showed that HFL found all those vulnerabilities, at least three times faster than Syzkaller. Moreover, each feature of HFL, addressing the challenge stemming from the kernel's unique characteristics, also showed substantial improvements.

[2]We will use *syscall* to indicate system call herein.

The key contributions of our paper can be summarized as follows.

- **The First Hybrid Kernel Fuzzer.** We propose a hybrid kernel fuzzing, HFL, leveraging the benefits from both random fuzz and symbolic testing techniques.
- **Handling Kernel-specific Challenges.** We identify three key kernel-specific challenges in applying hybrid fuzzing, and design HFL to resolve such challenges. In particular, we convert indirect control-flows to direct ones, making hybrid fuzzing more effective. We also keep the consistency of the kernel's internal states through inferring syscall sequence during the process of HFL. Further, we effectively reason about interfaces of syscall arguments, which are often in the form of complex and multi-layered structures.
- **Experimental Results in Bug Discovery and Code Coverage.** In our evaluation, HFL found 24 previously unknown vulnerabilities in recent Linux kernels. Additionally, HFL achieves around 15% and 26% higher code coverage over Moonshine and Syzkaller, and over kAFL/S2E/TriforceAFL, it achieves even four times better coverage using the same amount of resource (CPU, time, etc.). In terms of vulnerability discovery performance, HFL found the known 13 vulnerabilities over three times faster than Syzkaller.

II. Background

To overcome the limitations of random testing, a number of recent studies [32, 34, 44, 51, 53] have been applying symbolic execution to complement fuzzing. Composing such a hybrid is popular based on the fact that traditional fuzzing and symbolic execution have a negative correlation as we describe in the following.

Traditional Fuzzing. *Traditional Fuzzing* generally refers to a technique that generates random input to test the target program. Because this random testing is performant and scalable, it can quickly test a vast number of real-world applications. In particular, representative fuzzers such as AFL [52], ClusterFuzz [21], and Syzkaller [46], have shown remarkable results discovering many software vulnerabilities. However, such random fuzzing techniques are often stuck due to the inherent limitation; testing with randomly generated input cannot explore program paths beyond a tight branch condition, *e.g.*, `if (i == 0xdeadbeef)`, as it requires to guess a single value out of a huge (2^{32}) space. This limitation can easily be overcome by symbolic execution as we describe next.

Symbolic Execution. *Symbolic Execution* is a program testing technique that can generate an input that drives the target program's execution to a certain program path. To do this, symbolic execution takes all input to the target program as symbols and keeps tracking the program's runtime context as symbolic expressions. For instance, when a symbolic execution meets a conditional branch, it will keep track of branch conditions as path constraints with respect to symbols. When

the execution reaches a program path of interest, symbolic execution can generate an input to drive the program to that path by solving symbolic constraints on symbols. A critical limitation of symbolic execution is that it suffers from the state explosion problem. This happens whenever the execution meets a conditional branch. More specifically, as the symbolic execution meets a conditional branch, it has to explore both sides of branches, doubling the number of paths (i.e., states) to explore after passing such a branch. Typical programs contain a vast number of conditional branches, and even worse, a loop that processes the input will make the number of to-be-explored paths grow far faster.

Hybrid Fuzzing. *Hybrid Fuzzing*, combining fuzzing and symbolic execution, can complement the aforementioned limitations of each approach. On one hand, random fuzzing is often blocked by tight branch conditions, however, symbolic execution can provide an input to explore such a branch. On the other hand, symbolic execution suffers from the state explosion problem, however, fuzzing can guide symbolic execution to only explore a specific path by using an input being tested by fuzzing (*i.e.*, concolic execution), avoiding the state explosion problem. Exploiting advantages of each approach, hybrid fuzzing typically achieves better code coverage than solely applying each technique. For example, hybrid fuzzers for user-level applications such as Driller [44] and QSYM [51], have been demonstrating outperforming results to typical fuzz testings [39]. However, to the best of our knowledge, we note that hybrid fuzzing has not been applied to the kernel.

III. Motivation

In this section, we highlight challenges in applying hybrid fuzzing to kernel as our motivation of designing HFL.

A. Challenges in Applying Hybrid Fuzzing to Kernel

Combining fuzzing and symbolic execution is a promising approach. However, achieving this in kernel fuzzing is challenging. We found the following three challenges towards employing hybrid fuzzing in the kernel:

1) Indirect control transfer determined by input,
2) Internal system state requirements, and
3) Nested argument type inference.

These challenges render naive hybrid fuzzing, which is a simple integration of fuzzing and symbolic execution, does not work well in the kernel. More importantly, they stem from the unique characteristics in the kernel.

Before we elaborate each challenge in detail throughout this section, we provide a summary of how those challenges make both fuzzing and symbolic execution difficult as follows. First, kernel uses lots of function pointers to support polymorphism as a hardware abstraction layer. However, the use of function pointer makes both fuzzing and symbolic execution inefficient, and thereby, naive hybrid fuzzing also becomes inefficient. Second, kernel execution often depends on a specific internal state, but typically neither fuzzing nor symbolic execution handles this, and failure to building such a state will render testing counterproductive. Building an internal state for having meaningful testing requires analysis in system call dependencies (e.g., calling order, arguments, etc.), but hybrid fuzzing that does not handle this can match the state only by a luck, which will waste many testing executions. Third, some system calls require their parameters to hold nested data structures, and not analyzing such nested structures makes both fuzzing and symbolic execution unable to generate test programs for exploring kernel execution that depends on the nested structure. Testing trials without considering these challenges will not result in a meaningful kernel execution thus are not helpful for testing progress.

Unsolved Challenges in Kernel Testing. Those challenges are not only specific to hybrid fuzzing. They also render techniques that solely runs either fuzzing or symbolic execution inefficient. Nonetheless, most of them are not handled well in fuzzing literature. Table I lists the characteristics of recent kernel testing methods. Techniques used in the first six fuzzers, such as perf_fuzzer [49], Digtool [35], kAFL [40], Razzer [26], PeriScope [43], and FIRM-AFL [54], do not handle the aforementioned kernel-specific challenges. CAB-Fuzz [28], which is an S2E-based symbolic execution fuzzer, handles strict kernel branch conditions, but it does not handle indirect branches nor the rest of the challenges. Regarding the second challenge, inferring system call sequence for building internal system state, IMF [24] attempts to resolve this by analyzing system call dependencies using syscall traces. However, their analysis is based on example traces, which is ad-hoc, and can only infer type and argument dependencies. MoonShine [33] digs further in this challenge by analyzing system call dependencies via static analysis. However, such a static approach generates many false positives, and it does not infer dependencies in the "value" of parameters, which are only available at runtime. Regarding the third challenge, inferring nested argument types, DIFUZE [16] applies static analysis to infer types of complex syscall arguments. Unfortunately, many of types used in the kernel are defined as abstract pointers (e.g., `void *` or `unsigned char *`). Because the exact type information (e.g., size, type flags, etc.) is only available at runtime in such a case, such a static approach cannot precisely determine the type of nested objects.

To the best of our knowledge, there is no prior work that handles kernel-specific fuzzing challenges at runtime, which is essential to enable an efficient kernel hybrid fuzzing. In the following, we elaborate on the challenges in detail with examples to specifically demonstrate why such examples block both fuzzing and symbolic execution in exploring kernel code.

B. Indirect Control Transfer Determined by Input

The polymorphism pattern in Linux kernel, which often transfers the kernel control flow via function pointer tables accessed by system call arguments, makes applying traditional testing techniques to kernels difficult.

Challenge 1. Discovering Indirect Control-Flow. Linux kernel makes heavy use of a function pointer table accessed by system call parameters, tightly related to its design philosophy. To support a huge number of different devices or features, i.e., supporting polymorphism with a single interface, most components in Linux are decoupled with its abstract interface and implementation layer, where the interface layer is generically used for accessing a specific implementation. This in fact is similar to employing polymorphism commonly exercised

TABLE I: The comparison of recent kernel fuzzing techniques.

Technique	Target Kernel	General Requirements		Kernel-specific Requirements			
		State Explosion Free	Coverage Guided	Handling Strict Branch Condition (Naive hybrid fuzzing)	Handling Indirect-Control Flow (§III-B)	Calling Sequence Inference (§III-C)	Nested Syscall Argument Retrieval (§III-D)
perf_fuzzer [49]	Linux (perf_event set)	✓	×	×	×	×	×
Digtool [35]	Windows	✓	×	×	×	×	×
kAFL [40]	Win/Linux/macOS	✓	✓	×	×	×	×
Razzer [26]	Linux	✓	×	×	×	×	×
PeriScope [43]	Linux (drivers)	✓	✓	×	×	×	×
FIRM-AFL [54]	Firmware	✓	✓	×	×	×	×
CAB-Fuzz [28]	Windows (drivers)	×	×	✓	×	×	×
IMF [24]	macOS	✓	×	×	×	✓	×
MoonShine [33]	Linux	✓	✓	×	×	✓	×
DIFUZE [16]	Android	✓	×	△	×	×	✓
HFL	Linux	✓	✓	✓(§IV-A)	✓(§IV-B)	✓(§IV-C)	✓(§IV-D)

in object-oriented programming languages, such as C++ and Java, and Linux accommodates such a concept by using a function pointer table in the C language. More specifically, Linux typically constructs a function pointer table (i.e., an abstract interface), which contains a list of function pointers pointing to concrete implementation. When the kernel performs a specific operation at runtime, it fetches a corresponding function pointer by indexing the table. Such a use of a function pointer table, which are heavily occurring in Linux kernel implementation, severely hinders traditional testing schemes from extending code coverage.

Example 1: Indirect Control-Flow in RDMA/AUTOFS. The example shown in Figure 1 illustrates a case of having an indirect control transfer caused by using a function pointer table in the driver, where data communication over Remote Direct Memory Access (RDMA) network is managed/operated. In this example, `ucma_write()` uses a function pointer table, namely `ucma_table`, by indexing the table with the header information of data, controlled by user-level input, `char __user *buf`. The function table, `ucma_table`, holds an array of function pointers (lines 1-8), where each function pointer implements a specific functionality of network communication, such as `connect` and `bind`. The functions assigned in the array are indirectly invoked at line 16. In particular, a value from userspace, `hdr.cmd` (copied from `buf`) serves as an index to the function table. Depending on the index value, different functions will be executed from the table. Autofs is another case in point, a service program for automatically mounting various file systems. In Figure 2, `autofs_ioctl` acts as a dispatcher, which invokes various underlying control functions, using a function pointer table `_ioctls`. In a similar way, `cmd` derived from the userspace implicitly affects the following control-flow transfer via an indirect function call.

Although it may be easy to manually understand this code snippet, automatically exploring such a control transfer is challenging for conventional testing techniques. In the case of traditional fuzzing, it has to correctly guess all the array index values to explore all target functions stored in a function pointer table. However, matching such concrete index values based on random mutation would be like finding a needle in a haystack because the probability of hitting the correct index value by a chance is extremely low (e.g., 23 correct function indices among 2^{32} possible values).

```
1  ssize_t (*ucma_table[])(struct ucma_file *file,
2      char __user *inbuf, int in_len, int out_len) = {
3    [RDMA_CREATE_ID]      = ucma_create_id,
4    [RDMA_DESTROY_ID]     = ucma_destroy_id,
5    [RDMA_BIND_IP]    = ucma_bind_ip,
6    ...
7    [RDMA_JOIN_MCAST]     = ucma_join_multicast
8  };
9  ssize_t ucma_write(struct file *filp, char __user *buf,
10     size_t len, loff_t *pos) {
11   struct rdma_ucm_cmd_hdr hdr;
12   ...
13   if (copy_from_user(&hdr, buf, sizeof(hdr)))
14   ...
15   // indirect function invocation
16   ret = ucma_table[hdr.cmd](file, buf + sizeof(hdr), hdr.in, hdr.out);
17 }
```

Fig. 1: A simplified example with respect to indirect control-flow.

In the case of symbolic execution, if it is capable of a symbolic pointer dereference [10], in theory, it does not suffer from exploring such a case. Specifically, when it faces a symbolic dereference accessing the function pointer table (e.g., `func_ptr[symbol]()` in Figure 1), it can symbolically dereference a value pointed by a function pointer (i.e., retrieve a function pointer within a function pointer table). However, allowing such a symbolic dereference would suffer from state explosion issues. This is because a symbolic execution technique is not aware of whether a dereference operation is fetching from the code or data pointer, so it attempts to perform the symbolic dereference for all dereference cases (including dereferences for both function pointer table and non-function pointer table). As a result, it performs a dereference for a non-function pointer table as well, which is unlikely increasing the code coverage. Thus, it would introduce a huge number of trials (i.e., attempting to dereference using all indices of non-function pointer table) and each trial requires new path exploration, resulting in state explosion issues.

In order to address this issue, HFL transforms such a function pointer table dereference into a more explicit form of control-flow transfer. So HFL instructs the symbolic execution engine to prioritize on cases dereferencing function pointer table and de-prioritize non-function pointer cases.

```
1  static int autofs_ioctl(unsigned int cmd,
2     struct autofs_dev_ioctl __user *user) {
3    static ioctl_fn _ioctls[] = {
4        autofs_ioctl_version,
5        autofs_ioctl_protover,
6        autofs_ioctl_protosubver,
7          ...
8        autofs_ioctl_ismountpoint,
9    };
10   unsigned int idx = cmd_idx(cmd);
11   ...
12   fn = _ioctls[idx];
13   ...
14   // indirect function invocation
15   fn(fp, sbi, param);
16 }
```

Fig. 2: A code snippet of an indirect function call through a function pointer table.

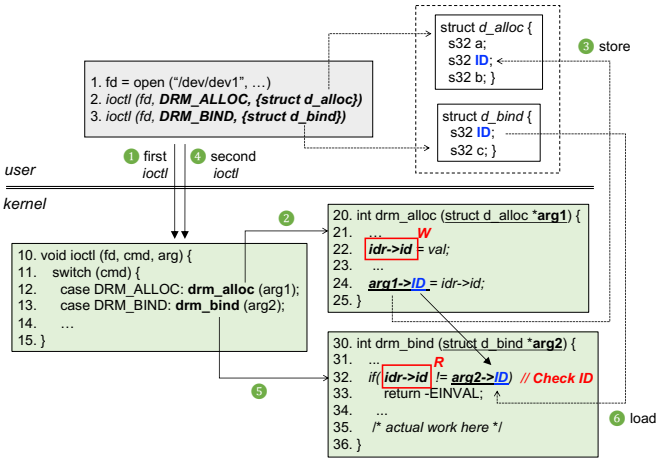


Fig. 3: The requirement for consistent kernel state.

C. Internal System States

Challenge 2: Coordinating Internal System States. A kernel maintains internal system states to manage computing resources. In particular, it keeps track of per-process contexts (i.e., virtual memory, file descriptors, etc.) or manages peripheral devices to enable shared accesses to those. Most of these internal system states are transitioned mainly through a syscall, because a syscall is a key interface that the kernel accepts the commands from the user space. Since this syscall and the system states are highly correlated, if a syscall is invoked without setting up the system states properly, the syscall would be simply rejected by the kernel.

For this reason, traditional fuzzing is limited in handling the kernel as it cannot cater this internal system state well. More specifically, because it randomly constructs a syscall, it cannot figure out intricate rules behind syscalls to properly setup internal system states — how to order multiple syscalls, parameter dependencies between those.

Concolic execution[3] does not work well either. As mentioned before, an important decision in performing concolic execution is to minimize the number of symbolized data variables, otherwise it would suffer from state explosion issues. However, there are a huge number of such data variables in the kernel (virtually all global/heap variables may look-alike related to system states).

As we will explain later, HFL addresses this issue through performing points-to analysis, which guides our concolic execution to selectively symbolize data variables per-syscall, rendering HFL's concolic execution interpret internal system states well.

Example 2-1: Obvious Syscall Sequence. In order to provide file system access, the kernel maintains a file descriptor per file where its state can be shortly defined as { opened, closed }. Specifically, in response to `open` syscall, a new file descriptor is returned where its kernel internal state is initialized as opened. Then following `read/write` syscalls are only working if the state of the file descriptor (specified within read/write syscalls) is opened. Therefore, read/write syscalls should be invoked only after the open syscall. Otherwise, read/write syscalls would immediately return an error, limiting the further code coverage exploration.

We observe that kernel imposes much more complex syscall orders than this simple example. They often accompany internal argument types behind syscall interface, which are implicit in system call definition, as follows.

Example 2-2. Complex Syscall Sequence in DRM. Direct Rendering Manager (DRM) is responsible for managing graphic devices and memory in the Linux system. Figure 3 illustrates a simplified example of `ioctl` syscall dependency in the DRM driver. In the example, if the `cmd` value is `DRM_ALLOC`, the kernel executes `drm_alloc`. It initializes `ID`, which in turn returns back to the user program through the `ID` field (line 24). This initialized `ID` value will be the anchor for the next syscall accessing DRM devices, in which the `ID` value refers to the previously `DRM_ALLOC`-ed DRM device. One of such follow-up use cases is using the `cmd` value as `DRM_BIND`. In this case, the kernel executes `drm_bind` and the `ID` value is passed to check for consistency (line 32).

To enhance the coverage in the example, we should consider two basic conditions. First, we maintain the calling order of syscalls (i.e., invoking two `ioctls` where the first is using cmd `DRM_ALLOC` and the second is using cmd `DRM_BIND`). Next, when invoking these two syscalls, the syscall parameter dependency should be kept as well. In other words, the `ID` field specified through the `arg` parameter should be propagated—when invoking cmd `DRM_BIND`, its `ID` field should be using the `ID` value returned from cmd `DRM_ALLOC`.

A recent study, IMF [24] attempts to infer kernel system internals by modeling system call sequence. Given syscall traces and argument/return types, it implicitly models internal system by tracking value-flows across syscall arguments and returns. However, because of its limited analysis scope to user-domain, IMF is unable to reason about dependencies within the kernel (e.g., `idr->id`), and fails to track the flows of the arguments whose types are invisible in user space (e.g., `struct d_alloc`). Such an approach specific to user-domain makes the inference of kernel state incomplete, thus hinders deeper kernel code exploration in the end.

Moonshine [33] constructs dependency pairs through static analysis, and learns internal state dependencies. However, since it only relies on points-to analysis, its acquired knowledge is

[3]This paper uses symbolic execution and concolic execution interchangeably.

limited to the dependencies between system state variables (i.e., variable `a` is aliased with another variable `b`, or the value of `a` is derived from `b`). Compared to Moonshine, HFL performs both point-to analysis and symbolic checking, so it can also figure out precise constraints between those (i.e., variable `a` should be the same as variable `b`, or variable `a` should be the same as the addition of variable `b` and `c`), which significantly augments code exploration capability of HFL (IV-C).

D. Nested Syscall Arguments

Challenge 3. Constructing Nested Syscall Arguments. The kernel is designed to copy data from/to user-space, as it has to take/return data from/to user-space to serve syscalls. This copy operation is always performed through specific kernel APIs, such as `copy_from_user` and `copy_to_user`, as the kernel cannot directly access user-space memory for security reasons, i.e., Supervisor Mode Access Prevention (SMAP) or Kernel Page Table Isolation (KPTI). Specifically, `copy_from_user` copies a block of data from user-space into a kernel buffer while `copy_to_user` does so from kernel to user.

We observe that syscall arguments are often constructed as nested structures (i.e., a field member in one structure points to another structure), where this nested feature is being supported by the above-mentioned `copy_from_user`. More importantly, the precise layout of such nested structures can only be known at runtime in many cases: the structure usually has a size variable indicating the size of the next nested structure so as to minimize the size of the to-be copied data.

To better illustrate how these nested structures are supported by the kernel, the following is the common case how the kernel handles a nested structure: 1) the kernel first takes a pointer (specified as a syscall argument) pointing to a data structure `A` located in the user-space; 2) the kernel dynamically allocates the buffer (within the kernel) to hold the copy of `A`; 3) the kernel copies `A` from user-space to this allocated buffer using `copy_from_user`; 4) referring to a size parameter within `A` (which indicates the size of a nested structure `B`), the kernel allocates another buffer to hold the copy of `B`; 5) the kernel performs another `copy_from_user` to copy `B` from user-space to its allocated kernel buffer.

Without prior knowledge about such a nested form of argument structures, traditional fuzzing is unable to fuzz the entire argument structure properly as it hardly figures out complex argument formats behind syscalls. Likewise, symbolic execution fails to infer precise nested structure because it is only aware of given input space explicitly symbolized; in other words, it cannot propagate symbolizations beyond nested memory buffers connected through pointer variables.

Example 3. Nested Syscall Arguments in USBMon. Figure 4 exhibits a simplified function `proc_control`, which basically controls USB devices connected. The function, derived from `ioctl` syscall, requires multi-layered memory buffers which are initially indicated by its second argument (i.e., `__user arg`). At line 9, the outer memory buffer is first copied into allocated kernel buffer `ctrl`. Returned without an error, a subsequent memory copy, pointed by a field member of the previously copied buffer (*i.e.*, `ctrl.data`), occurs with a certain size (*i.e.*, `ctrl.len`) at line 11. Note here that without built-in knowledge of such a nested and variable-sized form of syscall parameters,

```
1. struct usbdev_ctrl {
2.   void __user *data;   // unknown type
3.   unsigned len;
4. }
5. int proc_control (…, void __user *arg) {
6.   struct usbdev_ctrl ctrl;
7.   unsigned char *tbuf;   // unknown type
8.   …
9.   if (copy_from_user (&ctrl, arg, sizeof(ctrl))
10.  …
11.  if (copy_from_user (tbuf, ctrl.data, ctrl.len))
12.  …
13.  /* main usb control communication */
14.  …
15. }
```

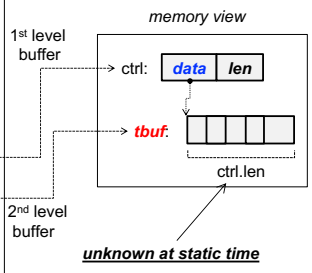


Fig. 4: An example presenting nested syscall arguments.

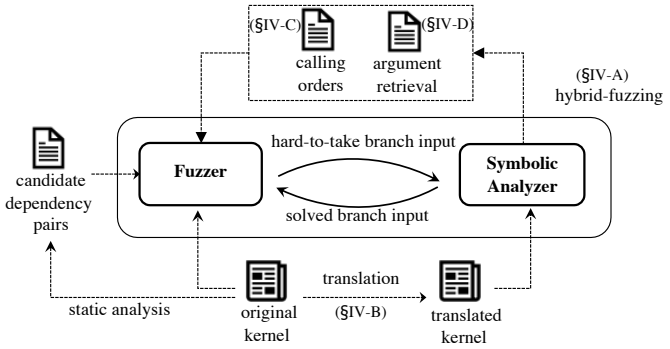


Fig. 5: Overview of HFL.

the execution likely stops (at either line 9 or 11) due to invalid memory access, before reaching its main functionality (line 14).

As shown in Table I, DIFUZE [16] is capable of identifying complex syscall arguments. With regards to a nested form of argument structure, its strategy is to statically keep track and infer the type of pointer fields in the structure which determines the inner structure type. Unfortunately, such a static approach is limited in understanding the nested structure because in the example above, the type of the inner buffer is not statically determined (i.e., `unsigned char *`), and the size of it is resolved at runtime (i.e., `ctrl.len`).

IV. Design

This section describes the design of HFL. At a high level, HFL takes a hybrid fuzzing approach, combining traditional fuzzing and concolic execution techniques (§IV-A). Moreover, in order to address the kernel-specific challenges in performing hybrid fuzzing, HFL designs three different features: 1) converting indirect control-flows to direct ones (§IV-B); 2) consistent system state via calling sequence inference (§IV-C); and 3) retrieving nested syscall arguments (§IV-D).

A. Generic Hybrid Fuzzing Design

Overall, the design of HFL follows previous user-level hybrid fuzzing techniques, which combine traditional fuzzing and symbolic execution, as shown in Figure 5. HFL's general fuzzing features can be characterized with the following three features: 1) kernel syscall fuzzing, 2) coverage-guided fuzzing, and 3) symbolic analysis.

Kernel Syscall Fuzzing. HFL identifies kernel bugs that are triggered by a user program. As such, HFL's fuzzing scheme focuses on generating or mutating a user program, which is a sequence of syscalls, similar to how Syzkaller [46] works. In order to respect the rules in invoking syscalls, HFL constructs (or mutates) syscalls based on pre-defined syscall templates. This pre-defined syscall template dictates how the syscall has to be constructed. The template includes a list of available syscalls and its format. For each syscall, the template instructs the type of syscall parameters, a range of constant values of syscall parameters, dependency between syscalls (i.e., a return value of some syscalls should be used as a parameter of other syscalls).

We note that Linux does not provide a formally defined rule for syscall invocation, thus this pre-defined syscall template is not only manually defined but also incomplete, as we evaluated in §VI-C1. This in fact motivated HFL to design fully automated kernel-specific features, described throughout this section.

Coverage Guided Fuzzing. HFL follows a generic coverage guided fuzzing scheme, as most other fuzzers do (including AFL [52] and Syzkaller [46]), which prioritizes its fuzzing input mutation strategy towards extending the execution coverage. HFL instruments all code blocks in the kernel, and collect execution coverage information when executing a user program. Based on such execution coverage information, HFL can determine whether a new input (i.e., a newly generated or mutated user program) assists to augment the execution coverage. To achieve so, HFL keeps track of all code blocks that are covered before during fuzzing, and checks if the new input executes any uncovered block. If so, HFL pushes the new input into a corpus (i.e., a set of inputs, each of which will be mutated later).

Symbolic Analyzer. Fuzzer's well-known limitation is that it cannot drive an execution passing through a hard constraint imposed in some branch conditions. Thus, fuzzer's execution coverage does not improve once reaching branches with those hard constraints. This in fact motivates the adoption of hybrid fuzzing techniques, which typically combines symbolic execution to solve such hard constraints.

More specifically, during the fuzz testing, HFL's fuzzer identifies a hard-branch in the kernel, which was always evaluated into true (or false) throughout a number of user program executions, and thus the code reachable with the other evaluation result was never explored. To detect such hard-branches, HFL maintains a frequency table during the fuzz testing, counting the number of true/false evaluations per branch; thus it is able to filter out uninteresting branches whose underlying blocks are already seen. Once identified, one of the corresponding user program triggering the hard branch is passed over to the symbolic analyzer. Then the symbolic analyzer symbolizes all the syscall parameters in the given user program, and then starts performing typical concolic execution until it reaches the hard branch. After reaching, the symbolic analyzer queries the solver if it can find a symbolic assignment flipping hard-branch's evaluation results. If so, based on the solved symbolic assignment, the symbolic analyzer reconstructs the user program which flips the evaluation of the hard-branch, providing a new user program to the fuzzer. It is worth noting that, since path exploration is offloaded to HFL's fuzzer, the symbolic analyzer is limited to following along a single execution path (path explosion-free) and on-demand constraint solving.

```
// before translation
ret = ucma_table[hdr.cmd](...);

// after translation
if (hdr.cmd == RDMA_CREATE_ID)
  ret = ucma_create_id (...);
else if (hdr.cmd == RDMA_DESTROY_ID)
  ret = ucma_destroy_id (...);
...
```

Fig. 6: Conversion to direct control-flow.

B. Converting Control-Flow from Indirect to Direct

As described before in §III-B, heavy use of a function pointer table in the kernel makes hybrid-fuzzing ineffective, mainly because it introduces indirect control-flows that are unfriendly to traditional analysis techniques.

To this end, HFL designs *an offline translator*, operating based on the source code of the kernel, which transforms indirect control-flows to direct control-flows. The translator unfolds an indirect control-flow to a direct one while maintaining the semantics of conditional branches, such that all underlying code blocks are reachable through direct control-flows. Specifically, the translator iterates over each instruction at compile-time. When facing the indirect control-flow, the following procedure is performed: 1) The translator ensures that an index variable of the function pointer table originates from syscall parameters. This is because we are not interested in control transfer patterns, which are not controllable by syscall parameter mutation. Thus, HFL keeps track of how syscall parameters are propagated by performing inter-procedural data-flow analysis. Considering speed/accuracy trade-off, our data-flow analysis is context-, and flow-insensitive but field-sensitive. 2) Given the function table and its feasible index values, HFL performs branch transformation (similar to loop unrolling) — for each index value, HFL inserts a conditional branch jumping to a corresponding function pointer. Figure 6 summarizes the simplified result of the code transformation.

C. Consistent System State via Syscall Sequence Inference

Random testing in charge of executing a sequence of system calls often fails to explore much of kernel code but returns an error early (e.g., return with `-EINVAL;`). This is because if syscall execution sequences are not following its intended semantics, the kernel's internal states are not accordingly setup to perform syscalls (§III-C).

In order to address this issue, HFL infers a proper order of syscalls and syscall dependency. To this end, HFL first obtains potential dependency pairs, as a result of static analysis on the kernel, then validates the collected dependencies to distinguish true dependency pairs. Further, it detects parameter value dependencies by keeping track of dependency value propagation connected with symbolized syscall arguments. Once a valid order of syscall sequences are retrieved, HFL provides feedback for fuzzer such that it can be immediately applied for future mutation. In the following, we describe each step in detail.

1) Static Analysis to Find Candidate Dependency Pairs. As a first step, HFL performs static analysis to capture candidate dependency pairs. In particular, HFL performs inter-procedural points-to analysis on the target kernel, collecting a pair of read/write operations, i.e., one instruction performs the read instruction and the other performs the write, where both instructions are reading from and writing to the same memory location. We call these read/write operation pairs as *candidate dependency pairs*. Note that this analysis is performed offline before performing hybrid fuzzing, and the next phase, which is part of hybrid fuzzing, takes such candidate dependency pairs as input.

2) Runtime Validation to Identify True Dependencies. Given a set of candidate dependency pairs, HFL now starts concolically executing the kernel. In order to filter out false dependencies due to the inherent false positive issues of the points-to analysis, HFL performs basic validation in this phase. More specifically, when HFL symbolically executes both instructions of any candidate dependency pair, HFL checks if these access the same address. If so, it indicates that instructions in this dependency pair truly depend on each other, yielding *a true dependency pair*. It is worth noting that once identifying this true dependency, HFL is able to infer syscall invocation order — i.e., the syscall performing the write operation has to be invoked before the syscall performing the read, because the write operation may initialize the value that the read operation relies on. If the write is not performed beforehand, the syscall including the read may simply return an error.

3) Symbolic Checking to Detect Parameter Dependency. Besides determining the order of syscalls, a dependency pair also determines multiple parameters across syscalls (shown in §III-C). To learn this, HFL makes use of symbolic constraint information, coming from symbolized syscall arguments. Specifically, HFL keeps track of the flow of the value caused by dependency objects, and figure out the out/inbound points of its read/write operation, respectively. Also, this allows to identify relevant memory location (offsets) and the size of it, out of symbolized argument memory in syscall.

Since HFL's fuzzer constantly interacts with the symbolic analyzer, HFL immediately feeds the output of the ordering set produced by concolic execution to the fuzzer, and reflects the up-to-date syscall-order information in the later mutation. In this manner, HFL keeps identifying new syscall relations and updating the result until its termination.

Example: Inferring Syscall Dependency to Reconstruct System State. Figure 7 depicts a workflow of our syscall sequence inference using the example in Figure 3. Given a user program along with (candidate) instruction dependencies, HFL starts executing the program concolically (❶). Meanwhile, the instructions belonging to the pairs are placed under observation. Once HFL hits the two instructions in a given pair (in any order) (❷), it examines the pair for runtime validation. Particularly, HFL takes two objects (i.e., operands) in the dependency relationship from the both instructions, then see if their memory addresses are equal (❸). If satisfied, this pair turns out to be a true dependency (❹).

Since the argument memory chunk in syscalls is symbolized at the initial phase (§IV-A), access to symbolized memory region propagated allows to locate an offset of dependency on the symbolic memory (❺). Once the execution is finished, it consolidates the true dependency all together (❻), HFL setups and establishes new syscall invocation rules (❼). As the last step, such a new set of invocation orders is fed back and will be used for mutating new input programs (❽).

D. Nested Syscall Argument Retrieval

Aside from syscall invocation sequences, system call fuzzing has to determine argument values. As shown in Figure 4, it is often required to understand complex nested argument structures, which are unknown in the syscall definitions, rendering fuzzer fail to keep exploring the kernel code.

Therefore, HFL understands and retrieves nested syscall arguments through a combination of concolic execution and kernel domain knowledge on data transfer functions. The reason why HFL focuses on the data transfer functions (e.g., `copy_from_user`) is that those are responsible for delivering data between user and kernel space, constituting the key mechanism in constructing the nested syscall arguments.

We observe that the following two pieces of information are the key to re-construct the nested syscall arguments: 1) memory location connecting to nested input structures; and 2) the length of memory buffer arguments.

To this end, we keep monitoring invocations of the transfer functions during concolic execution. Once invoked, we check if its source buffer is symbolically tainted. This allows concolic executor to focus on certain transfer functions that come from the syscall of interest. Thus we ensure the source buffer stems from the upper-level buffer originating from the parameter values. Similar to §IV-C, using its symbolic state, we keep track of a distance value (offset) to the location, where a pointer field (pointing inner buffer) will reside. Meanwhile, we can obtain the length of buffers by tracking parameter values of the transfer functions. This allows to learn nested buffers and the size of it as well.

Example: Retrieving Nested Syscall Argument. Figure 8 steps through the argument retrieval in detail. A syscall invocation reaches the internal kernel function `proc_control`, along with its argument values (❶). At the first invocation of the transfer function at line 4 (❷), we place the `arg` buffer under the control and obtain the buffer size (`0x14`) from its third parameter value (❸). Next, in the subsequent invocation at line 6 (❹), we make sure `ctrl.data` is symbolically tainted and both buffers are nested relation (❺). Using symbolic state of the buffer, we learn a corresponding offset value, where a pointer variable will later reside in the upper buffer. After the execution is terminated, we define a new argument rules for the invoked system call (❻-❼), then pass it over to fuzzer (❽).

V. Implementation

HFL is implemented on top of the existing fuzzing technique *Syzkaller* [46], and symbolic execution engine *S2E* [15]. We basically leverage the core features of both tools for basic fuzzing and symbolic execution features, and make significant adjustments to them to realize the design of HFL. For instance, HFL makes use of input generation algorithm equipped in Syzkaller as well as symbolic engine (e.g., constraint solver) from S2E. With respect to syscall argument symbolization, we

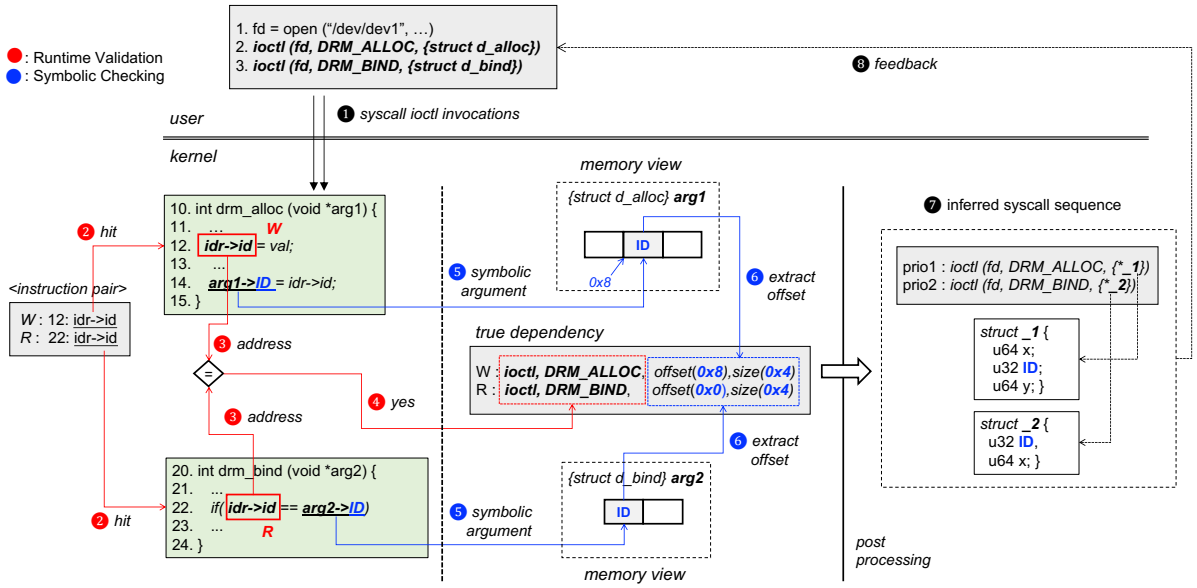


Fig. 7: Workflow of syscall sequence inference.

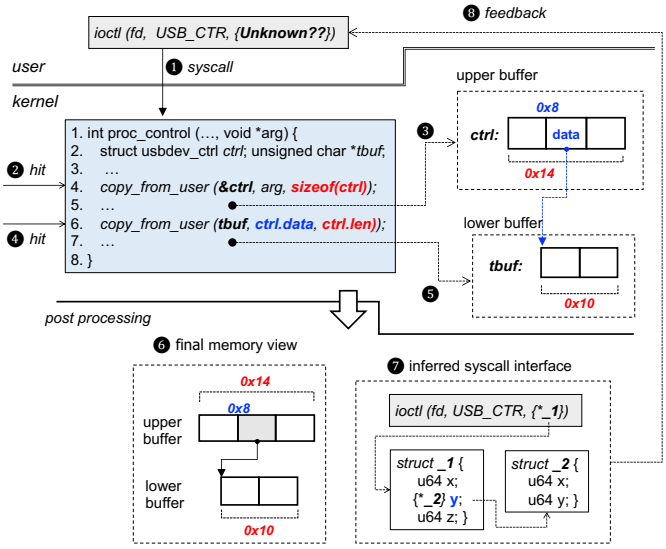


Fig. 8: Workflow of nested syscall argument retrieval.

leverage a function `s2e_make_symbolic` from S2E API, and adjust it to instrument the target kernel in many locations. To handle indirect control-flow, we augment *gcc* [3] (in particular, GIMPLE representation), to equip the translation functionality toward direct control-flow. To perform inter-procedural static analysis on the Linux kernel source code, we employ SVF static analyzer [45]. Since its analysis is carried out on LLVM IR, we translate the kernel source code to appropriate LLVM bitcode beforehand, using llvmlinux [18]. Table X summarizes our efforts of modifying the tools used in HFL. We open-sourced our reference implementation such that security analysts and researchers can benefit [1].

VI. Evaluation

In this section, we evaluate the effectiveness and efficiency of HFL. In particular, our evaluation examines both overall (§VI-B) and feature-specific (§VI-C) aspects of HFL. We aim to answer the following research questions:

- **Q1**: How effective is HFL in finding kernel bugs? (§VI-B1, §VI-B4)
- **Q2**: What is the overall coverage enhancement that HFL brings over existing approaches? (§VI-B3)
- **Q3**: How efficiently can HFL find bugs compared to other fuzzers? (§VI-B2)
- **Q4**: What is the contribution of each feature in HFL to the overall performance? (§VI-C1, §VI-C2)

A. Experimental Setup

All experiments are performed on a machine with an Intel Xeon E5-4655 2.50 GHz CPU and 512GB RAM running Ubuntu 14.04 LTS. For both fuzzer and symbolic analyzer, we make full use of 32 CPU cores[4] i.e., 16 dedicated virtual machines each (one core per VM). We use a 10 second timeout to prevent an unexpectedly time-consuming constraint solving from delaying the entire symbolic execution. For all experiments, obvious syscall invocation orders (e.g., `open-write`), which are simply retrieved by syscall definitions, are given by default, such that our evaluation can focus on testing the complex syscall sequences exclusively. We note that our evaluation follows a recent fuzzing evaluation work [29] although slightly different due to the unique features of OS-specific fuzzing.

Subsystem Classification. A kernel has a large codebase that many subsystems share together. Based on the functional characteristics of subsystems, relevance to different system call interfaces, and the likelihood of vulnerability [16, 33],

[4] For a fair comparison, all of baselines used in the subsequent experiments exercise 32 CPU cores as well.

Category	# Internal-type	Syscalls used
Device drivers	32	open, ioctl, write, read
Network	20	socket, accept, bind, listen, ioctl, getsockopt, setsockopt, sendto, recvfrom, sendmsg, recvmsg
File system	6	open, read, ioctl, write, lseek

TABLE II: Classification of subsystem for the experiment.

we classify the subsystems into three categories: *network*, *file system* and *device drivers*. Within each category, we examine a variety of implementations that handle important kernel features, chosen based on their availability and sustainability through different kernel versions. In order to find the system call interfaces used by the subsystems, we leverage the registered virtual file operations in the implementation code and identify the related system calls. Table II summarizes the result of the classification.

B. Overall Effectiveness

1) Vulnerabilities: Upon our implementation and test environment, we apply HFL to test several Linux kernels which were the latest at the time of the experiment. To detect vulnerabilities, we leverage known kernel-specific sanitizers[5]. Once configured, we run HFL and detect crashes being triggered. Table III shows the result of crash detection. In total, HFL found 51 vulnerabilities. We manually analyzed and figured out the root causes, distinguishing new vulnerabilities. In summary, *24 vulnerabilities turned out to be previously unknown*. We reported all the newly found vulnerabilities — out of them, 17 were confirmed (four of them are already patched by the respected kernel developers).

As shown in the table, of the unique crashes, many were triggered while fuzzing on, in particular, the kernel drivers. This is because, as we will describe in §VI-B3, the most coverage improvement lies in the device driver code, hence there exist more opportunities to discover crashes in this category. According to the crash type, most of them were caused by both *integer overflow* and *memory access violation*, followed by *uninitialized variable access*, etc. These reported crashes have security impacts, which can be abused to launch either Denial-of-Service(DoS) or arbitrary code execution attacks, negatively impacting the security of an entire operating system. Note that roughly half of the crashes were detected even in the stable version of the Linux kernels, rather than (unstable) release candidate (rc). Further, some of the crashes were discovered in the core part of the Linux kernel, such as *memory allocation* and *timer system*, impacting the entire kernel operations. It is worth noting that Linux kernel has become more mature over decades and has been exhaustively tested by tons of machines with high-performance computing capacity [46]. Nonetheless, we believe HFL showed the notable performance, in terms of bug finding capability.

2) Efficiency: Besides the discovery of vulnerabilities, we emphasize the superiority of HFL in terms of bug-finding performance. To compare with HFL, we extra run a random syscall fuzzer (using Syzkaller) in the same experiment environment as in §VI-B1. In such a limited experiment, we figured that both HFL and Syzkaller commonly discovered 13 known crashes

[5] kasan [5], kmsan [6] and ubsan [7]

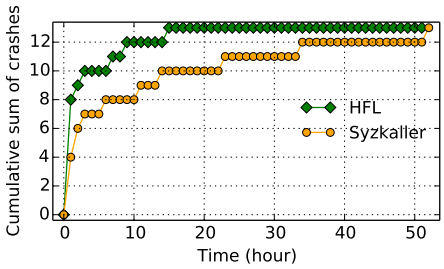


Fig. 9: Comparison of bug-finding time for 13 known crashes (Table IX).

(listed in Table IX) that were all existing crashes and already reported (but not patched) by other developers or analyzers. To compare the performance, we measured the time elapsed at the moment of each crash discovery, and then learn how fast HFL discovered all these crashes. Figure 9 summarizes the result of the performance comparison. As seen, HFL detected all these vulnerabilities at the earlier time (around 15 hours) than that of Syzkaller (over 50 hours), which reflects HFL shows better capability in its bug-finding efficacy.

3) Overall Coverage: The goal of HFL is to gain maximum code coverage by exploring as many execution paths as possible. To evaluate this, we count the number of unique code blocks that HFL has discovered throughout its entire process[6]. To highlight better results of HFL, we compare with 5 popular kernel fuzzers i.e., baseline Syzkaller, S2E [15], Moonshine [33], kAFL [40] and TriforceAFL[7] [22]. Since each of which has its own particular way to measure code coverage[8], we make all of them (including HFL) use the same coverage measurement, KCOV [4], for a fair comparison. For techniques that are not designed for directly testing various system calls, e.g., S2E and kAFL, we modified their frameworks to better handle such cases in our experimental environments for a fair evaluation. In the case that seed values are provided by default, we respect them to use in the test; otherwise we used a randomly generated one.

We then conduct experiments, taking a coverage measurement, for 50 hours 3 times. The result is summarized in Figure 10. In comparison, HFL not only reaches the peak point faster, but explores more code blocks for all the categories. For device drivers, in particular, the coverage difference between HFL and other techniques is huge over the other two. According to our analysis, it is mainly because of the frequent use of `ioctl` function, where its second argument `cmd` is hardly predictable and the third `arg` is typically unknown type featured with various-sized and multi-layered structure (III-D). We observed that Moonshine showed the most similar performance in terms of the coverage compared to HFL. However, we notice that its performance benefits would not go beyond that of HFL, because it cannot handle hard-to-take branches, which are

[6] Although our decision of coverage measurement may miss some of unique paths, considering overhead of measurement, we believe it is in a practical and effective way.

[7] AFL-based [52] Linux kernel fuzzer.

[8] HFL, Moonshine and Syzkaller exercise KCOV-based measurement whereas kAFL, S2E and TriforceAFL have measurement systems on their own [17, 22, 40].

Crash type	Description	Kernel	Subsystem	Status	Impact
integer overflow	undefined behaviour in mm/page_alloc.c	4.19-rc8	memory	patched	likely exploitable
integer overflow	undefined behaviour in net/can/bcm.c	4.19.13	network	patched	DoS
integer overflow	undefined behaviour in drivers/input/misc/uinput.c	4.19.13	drivers	patched	DoS
uninitialized variable access	undefined behaviour in fs/f2fs/extent_cache.c	5.0.7	file system	patched	DoS
memory access violation	use-after-free Read in ata_scsi_mode_select_xlat	4.17.19	drivers	confirmed	likely exploitable
memory access violation	use-after-free Read in raw_cmd_done	4.19-rc2	drivers	confirmed	likely exploitable
memory access violation	warning in pkt_setup_dev	4.19-rc2	drivers	confirmed	DoS
uninitialized variable access	uninit-value in selinux_socket_bind	4.19-rc8	network	confirmed	likely exploitable
memory access violation	undefined behaviour in drivers/block/floppy.c	4.19-rc8	drivers	confirmed	likely exploitable
integer overflow	undefined behaviour in drivers/net/ppp/ppp_generic.c	4.19-rc8	drivers	confirmed	DoS
uninitialized variable access	uninit-value in selinux_socket_connect_helper	4.19-rc8	network	confirmed	likely exploitable
integer overflow	undefined behaviour in ./include/linux/ktime.h	4.19.13	timer	confirmed	DoS
integer overflow	undefined behaviour in drivers/input/mousedev.c	4.20.0	drivers	confirmed	DoS
integer overflow	undefined behaviour in drivers/pps/pps.c	4.20.0	drivers	confirmed	DoS
memory access violation	general protection fault in spk_ttyio_ldisc_close	4.20.0	drivers	confirmed	likely exploitable
integer overflow	undefined behaviour in net/ipv4/ip_output.c	5.0-rc2	network	confirmed	DoS
integer overflow	undefined behaviour in drivers/scsi/sr_ioctl.c	5.0-rc2	drivers	confirmed	DoS
memory access violation	use-after-free Write in vgacon_scroll	4.17-rc3	drivers	reported	likely exploitable
memory access violation	use-after-free Write in do_con_write	4.17-rc3	drivers	reported	likely exploitable
task hang	task hung in drop_inmem_page	4.17.19	file system	reported	DoS
memory access violation	null-ptr-deref Write in complete	4.17.19	drivers	reported	DoS
integer overflow	undefined behaviour in fs/xfs/xfs_ioctl.c	4.19.19	file system	reported	DoS
memory access violation	undefined behaviour in fs/jfs/jfs_dmap.c	4.19.19	file system	reported	DoS
task hang	task hung in reiserfs_sync_fs	4.19.19	file system	reported	DoS

TABLE III: List of 24 previously unknown vulnerabilities in the Linux kernels discovered by HFL.

mostly covered by HFL's feature. Regarding S2E, it got stuck at an early stage as the number of symbolic states grows; thus much less code blocks were hit compared with HFL and the others. Both kAFL and TriforceAFL do not show impressive coverage results, because they neither resolve tight branch conditions nor consider OS-specific features stated in §III. In summary, the overall coverage improvement of HFL over Moonshine and Syzkaller was around 15% and 26%, respectively. Compared to kAFL, S2E and TriforceAFL, we observed HFL's coverage improvement was more than four times.

In this regard, Table IV highlights the percentage distribution of explored code blocks across the tested fuzzing schemes for the *overall* category at the termination of the 50-hour experiment. Note that unlike Figure 10, this comparison is based on the line number[9] due to incomparable code block addresses (which rely on its own kernel build environment). Most of the blocks explored by Moonshine and Syzkaller fall into the ones discovered by HFL. In terms of the number of blocks uniquely identified by each fuzzer, HFL exhibits much better performance (18.1%) against all the others, as a result of our hybrid approach along with kernel-specific solutions. Furthermore, we compared the upper bound number of the coverage, i.e., taking the upper bound as the total (absolute) code blocks statically obtained from the target Linux kernel[10]. With this maximum code block counts as the upper limit (100%) of the coverage, HFL has approximately explored 10.5% of the total coverage while exhibiting less percentages for the other baselines, shown in Table V. We think this result is due to HFL's design choice, which only mutates entry points of system calls. We hope to improve the coverage by extending our mutation scope to the other kernel input space, other than system calls, though it is out of scope in this paper.

[9] `addr2line` [2] is used for translation.

[10] Code block extraction is based on KCOV-assisted instrumentation.

	Coverage
$\{H \cap M \cap S\} - \{K \cup T \cup E\}$	38.6%
$H - \{M \cup S \cup K \cup T \cup E\}$	18.1%
$\{H \cap M\} - \{S \cup K \cup T \cup E\}$	12.1%
$\{H \cap M \cap S \cap K\} - \{T \cup E\}$	9.9%
$\{H \cap S\} - \{M \cup S \cup T \cup E\}$	4.1%
$\{H \cap M \cap S \cap K \cap T\} - E$	3.8%
etc.	13.3%
Union of all	100%

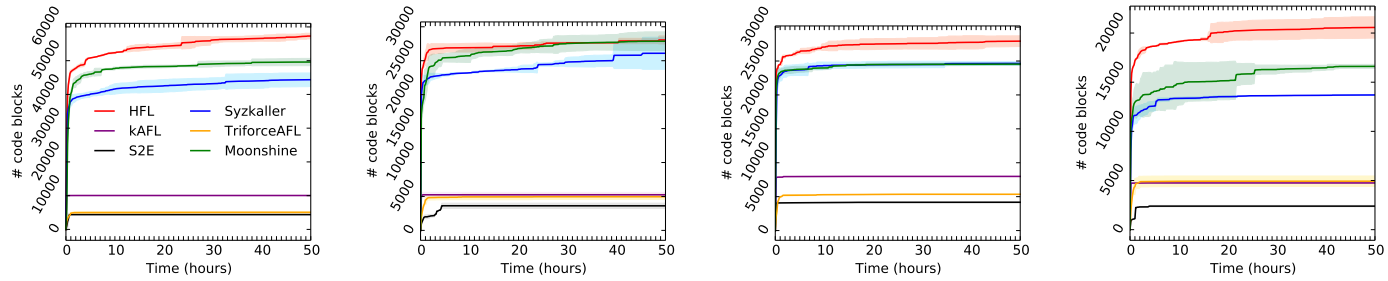
TABLE IV: Percentage distribution for the *overall* coverage result after running for 50 hours. 100 percent indicates the union of total code blocks found by all the baselines tested in the experiment. The notations **H**, **M**, **S**, **K**, **T** and **E** denote HFL, Moonshine, Syzkaller, kAFL, TriforceAFL and S2E, respectively.

	Coverage
HFL	10.5%
Moonshine	9.0%
Syzkaller	7.9%
kAFL	1.9%
TriforceAFL	0.9%
S2E	0.8%

TABLE V: Coverage percentage over the maximum (absolute) coverage. 100 percent coverage indicates the entire code blocks (statically identified in the target kernel binary) have been covered.

4) Case Study: In this section, we demonstrate a selective crash example discovered by HFL, and describe how HFL is used to reveal such a crash during the experiment.

page_alloc. Memory management is one of the key tasks of operating systems — through this subsystem, OS kernels govern the entire (physical and virtual) memory, and even the rest of subsystems rely on it for kernel memory allocation. In Figure 11, we present a simplified code snippet leading to a



(a) Overall **(b)** Network **(c)** File system **(d)** Drivers

Fig. 10: Coverage results during a 50-hours run. The line indicates averages while the shaded area represents 95% confidence intervals across three runs. The coverage improvement of HFL over Moonshine and Syzkaller is 15% and 26% (*overall*), respectively. Compared with the other three (kAFL, TriforceAFL and S2E), HFL shows more than 4 times of improvement.

crash in memory allocation procedure.

In the scenario, a user program opens a device file and conveys a message via syscall `ioctl`. Within the kernel, its corresponding implementation (i.e., `fd_ioctl`) sends a request for kernel pages, to store user-supplied data to be transferred. We noticed here that although the amount of the kernel pages requested is fully determined by the syscall argument, no sanity check was deployed along this particular path (against such data flow from argument input). Therefore, via syscall argument, illegitimate user input can immediately affect kernel page allocation, eventually leading to fatal memory problems, such as *out of memory*.

In order to follow such a bug-triggering path, we need to organize syscall arguments in a careful manner. At first, the variable `cmd` (derived from the second argument of the syscall) should be equal to a particular value (line 16). HFL precisely reasons about it by evaluating the expression relevant to the symbolized `cmd`. Second, it should have prior knowledge of an internal structure (i.e., `fd_raw_cmd`) of the input stream `param` (the third argument) such that `len` can be fuzzed enough to cause overflow (line 23); otherwise this field may remain unfuzzed. HFL keeps track of such a structure and correctly guesses the size of input stream, thanks to our syscall argument interface retrieval. On the other hand, other state-of-the-arts [24, 33] hardly reveal the vulnerability because they neither correctly resolve such a branch predicate nor infer its valid argument structure.

```
1  // user-supplied input format through ioctl function
2  struct fd_raw_cmd {
3     ...
4     void __user *data;
5     char *krn_data;
6     struct fd_raw_cmd *next;
7     long len;   // used for huge kernel memory request
8  };
9
10 int fd_ioctl(struct block_device *bdev, fmode_t mode,
11       int cmd, long param)
12 {
13    struct fd_raw_cmd *ptr;
14    ...
15    switch (cmd) {
16    case FDRAWCMD:   // tight branch condition
17    {
18       ptr = kmalloc(sizeof(struct fd_raw_cmd), GFP_KERNEL);
19       ret = copy_from_user(ptr, param, sizeof(*ptr));
20       ...
21       // BUG!! A large value of "ptr->len" can cause
22       //  out of memory in the following function
23       ptr->krn_data = (char *)fd_dma_mem_alloc(ptr->len);
24    ...
25 }
```

Fig. 11: Code snippet leading to a crash in kernel memory allocation

C. Per-feature Effectiveness

1) Per-feature Coverage: Since HFL is characterized with multi-featured fuzzing scheme, we verify how much each HFL feature contributes to the overall coverage. In order to better highlight, we conduct an extra experiment based on selective three test cases (i.e., `ext4`, `rds`[11] and `ppp`[12]), where each requires distinct HFL features to achieve the most performance improvement (Table VI). Considering the saturation points learned in our study, the experiment last up to 3 hours, we measure explored execution blocks afterward. To compare the result, we leverage existing pre-defined syscall templates (or rules) that are written by labor-intensive manual analysis [46]. Under the assumption that those template rules are well-crafted so as to cover all of the HFL's features, we serve it as the upper bound for this limited study despite its inherent scalability limitation.

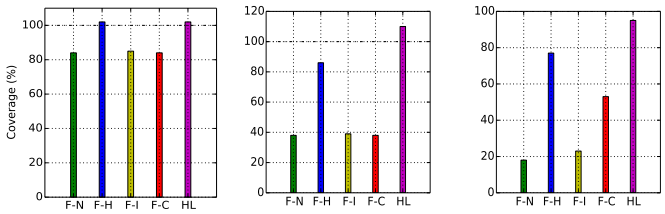
The experiment result is presented in Figure 12. Note that, in the figure, we represent both features, syscall sequence inference and argument interfaces retrieval, in a single line (**F-C**) because they are correlated i.e., depend on each other on their execution path. As expected, HFL equipped with all the features (**HL**) exhibits the high coverage improvement over the others, and its result is even close to that of the template-based case (100%). Interestingly, for the two cases (i.e., `ext4` and `rds`), we noticed ours even outperforms the template results (i.e., over 100%). According to our analysis, this is because their manual rules are either still in progress or not well-crafted enough. This apparently tells, unlike automatic and accurate nature of HFL, writing manual rules by human-effort is subject to error-prone as well as time-consuming task. Another notable observation is that the hybrid feature (**F-H**) makes a significant contribution to the overall coverage enhancement although other features

[11] Reliable Datagram Sockets (RDS).

[12] Point-to-Point Protocol (PPP). In this paper, PPP is considered as one of devices because it is accessed through a device file "`/dev/ppp`".

Testcase	Category	F-H	F-I	F-C
ppp	Drivers	✓		✓
ext4	File system	✓		
rds	Network	✓	✓	

TABLE VI: HFL's feature requirements for improving coverages with respect to selective 3 test cases. **F-H** and **F-I** denote the features of hybrid-approach and handling indirect control-flow, respectively. **F-C** represents a combined feature of both syscall sequence inference and argument interface retrieval.



(a) ext4 **(b)** rds **(c)** ppp

Fig. 12: Feature-specific coverage result for 3 test cases. **F-N** is a baseline fuzzer (absence of HFL features) while **HL** indicates HFL equipped with all of the features. For each test case, an appropriate system call rule is used for its upper limit (100%).

are still in demand. The reason is that on their execution path, other features heavily depend on the hybrid feature though hybrid feature alone barely reaches the maximum coverage. In other words, certain code blocks to be addressed by other features only appear when a strict branch condition, lying on the path at an earlier stage, is resolved by the hybrid feature.

In summary, we emphasize that all of HFL's features, rather than exclusively applying separate feature, are essential for accomplishing the maximum coverage. In addition, full featured HFL presents outstanding coverage improvement (even better than loosely written syscall templates) without ongoing manual efforts.

2) Per-feature Case Studies: In this section, using concrete examples found during the per-feature experiment above, we demonstrate and highlight the superiority of HFL in detail, compared with exiting kernel fuzzing schemes.

Figure 13 presents a piece of code in RDS network, where a control-flow transfer through function pointer table (line 11) is a practical obstacle against high execution coverage. As mentioned, random fuzz testing does not work well against such an indirect control transfer pattern. Observe that an extra operation on a variable flowing towards the array index (i.e., `optname - RDS_INFO_FIRST`) makes the fuzzer more difficult to hit all underlying function blocks. On the other hand, through the control-flow conversion, HFL correctly guesses and takes all elements of the array `rds_info_funcs`, thereby exploring underlying functions behind them.

In Figure 14, we showcase a simplified code, in which a calling sequence between two different `ppp` ioctls (`PPPNEWUNIT` and `PPPCONNECT`) is a prerequisite to operate correctly, leading to promoting coverage performance. Since IMF [33] relies on execution logs, such an internal dependency in the kernel

```
1  #define RDS_INFO_FIRST 10000
2
3  typedef void (*rds_info_func)(struct socket *sock, int len,
4      struct rds_info_iterator *iter, struct rds_info_lengths *lens);
5
6  rds_info_func rds_info_funcs[RDS_INFO_LAST - RDS_INFO_FIRST +1];
7
8  int rds_info_getsockopt(struct socket *sock, int optname,
9      char __user *optval, int __user *optlen)
10 {
11    func = rds_info_funcs[optname - RDS_INFO_FIRST];
12    ...
13    func(sock, len, &iter, &lens);
14    ...
15 }
```

Fig. 13: An indirect control-flow in RDS network.

```
1  long ppp_ioctl(struct file *file, int cmd, long arg) {
2     ...
3     switch (cmd) {
4     // 1. write dependency
5     // [syscall]: ioctl(fd, PPPIOCNEWUNIT, {VAL}<-unit)
6     // [NOTE]: VAL is written to untyped syscall argument
7     case PPPNEWUNIT:
8        // allocate an VAL to unit
9        err = ppp_create_interface(net, file, &unit);
10       if (err < 0)
11          break;
12       // write the VAL toward userspace
13       if (put_user(unit, arg))
14          break;
15       ...
16    // 2. read dependency
17    // [syscall]: ioctl(fd, PPPIOCCONNECT, {VAL}->unit)
18    // [NOTE]: VAL is read from untyped syscall argument
19    case PPPCONNECT:
20       // read VAL from userspace
21       if (get_user(unit, arg))
22          break;
23       ppp = ppp_find_unit(pn, unit);  // check the VAL for dependency
24       // [FAIL]: return if (untyped) value dependency is violated
25       if (!ppp)
26           goto out;
27
28       /* main connection procedure */
29       ...
30 }
```

Fig. 14: A piece of code involving a calling dependency across PPP ioctls.

is untraceable. Further, the corresponding argument types are too implicit to figure out in user domain. Moonshine [33] is capable of learning this dependency of the kernel object (i.e., `unit`) through offline static analysis on the kernel. Unlike HFL, however, it still suffers because they overlook the value flow caused by the dependency, going to/coming from syscall arguments (line 13 and 21). As a result, the execution likely terminates at line 25 due to inconsistent value (i.e., `VAL`) between the two `ioctl`'s arguments.

A code snippet in Figure 15 assumes that a syscall argument `arg` is structured in a nested format whose size is determined by `data.len` at runtime. To reach the deepest code path, HFL identifies such a nested format by observing the function invocation at line 11 and 15, and successfully constructs the required structure with a valid size. Although DIFUZE [16] attempts to retrieve such nested structure using static type inference, it eventually fails as the type of the pointer member remains unknown at static time (line 2).

```
struct ppp_option_data {
  __u8 __user *ptr;  // unknown typed inner buffer
  __u32 len;
  int transmit;
};
int ppp_set_compress(struct ppp *ppp, unsigned long arg)
{
   struct ppp_option_data data;

   // inferred buffer layout: {..|..|..}
   if (copy_from_user(&data, (void __user *)arg, sizeof(data)))
      goto out;
   ...
   // inferred buffer layout: {..|..|ptr} --> {...}
   if (copy_from_user(ccp_opt, (void __user *)data.ptr, data.len))
      goto out;
   ...
}
```

Fig. 15: Two-layered nested argument in PPP driver.

VII. Related Work

Traditional Kernel Fuzzing. For software testing, numerous research projects have been working on random fuzz testing due to its efficiency and effectiveness [22, 26, 36, 38, 44, 52]. Trinity [27] and Syzkaller [46] are popular coverage-guided system call fuzzers, targeting the Linux kernel. A recent kernel fuzzer, IMF [24] aims to infer kernel system states by keeping track of system call traces along with type information. Such syscall trace-based inference has limitations in understanding the true dependencies inside the kernel. Taking one step further, Moonshine [33] is particularly focusing on retrieving internal kernel dependencies through static analysis. However, such statically collected traces suffer from false-positive issues, so it is difficult to reason about internal system dependencies correctly. Compared to these, HFL is able to distinguish true dependencies through the validation during the kernel execution. DIFUZE [16] employs static analysis to effectively fuzz device drivers in the Android kernel. Although it is effective in a specific domain, it is unable to be applied to dynamically loaded modules and challenging to generalize. kAFL [40] aims to support a fuzzing framework targeting various (including closed-source) kernels, particularly supporting a hardware-assisted code coverage measurement. HFL, on the other hand, neither relies on imprecise static analysis results nor requires the hardware support. Periscope [43] devises a device driver fuzzer that is not related to system calls, but mutating input space over I/O bus. Razzer [26] designs a combination scheme of static and dynamic testing to effectively reach the points where race bugs potentially occurs, then verifying true bugs. Unlike HFL, however, those are difficult to be generalized due to manual efforts for source code modification or limited scope of analysis.

Symbolic Testing. There also have been a large body of researches on symbolic execution [8, 11, 12, 14, 15, 20, 23, 37, 41, 42]. In particular, most of which try to overcome its known weakness, *state explosion*. CAB-Fuzz [28] is designed to use per-function concolic execution and edge case priority to make it scalable. Similarly, UC-KLEE [37] performs symbolic execution on individual functions rather than program entry points, and filters out falsely catched crashes in a certain manner. However, their effort to recover pre-context using real app execution is limited to certain cases, and difficult to be generalized whereas performing post-process validation seems to work fine, but still yields false results.

Hybrid Fuzzing. Given their properties of fuzzing and symbolic testing, some of research have studied to benefit from both of them [32, 34, 44, 51, 53]. The early work of which [32, 34, 44] introduces and demonstrates such a powerful combination of the two schemes, highlighting its good performance in bug-finding and code coverage. More recent work places their focus on building more efficient hybrid fuzzing. In particular, they strive to maximize the performance of expensive symbolic engine [51] or boost symbolic execution utilization by stoping it from being idle [53]. In fact, all of them share the same insight as HFL in terms of collaboration of random and symbolic testing. The key difference is such hybrid-based work focuses only on application level testing, thus they cannot address kernel specific challenges, which completely hinder in-depth code block exploration in the kernel. With respect to exploring narrow-condition branches, laf-intel [25] is yet another method to tackle them. Without the help of the symbolic checking, it is capable of satisfying such a branch condition by breaking it down into multiple smaller branches, each of which is relatively easy to explore. We believe laf-intel is complementary to HFL, as each can benefit another in terms of solving narrow-condition branches.

Static Analysis for OS Kernels. Static analysis techniques have been extensively used to discover various types of vulnerabilities in kernels [19, 31, 47, 48, 50]. Dr.Checker [31] achieves soundy static analysis using the inter-procedural approach in a limited set of kernel drivers. K-miner [19] partitions the entire Linux kernel based on system call entry points, and analyzes the partitioned components separately. By tracking the flow of the code, it detects vulnerabilities with multiple analysis passes. However, unlike HFL, such a partitioned analysis does not fully understand relationships between system calls, making global analysis difficult. Research work [47, 50] statically analyze the code to find double-fetch bugs in the Linux kernel, whereas LRSan [48] specifically checks security-sensitive variables that can be modified after the first security checking. Compared with HFL, all of these techniques suffer from false positives, which would require manual efforts to identify true positive ones.

VIII. Discussion and Limitations

Kernel's Non-deterministic Behavior in Performing Symbolic Analysis. In our study, we observed non-deterministic program behaviors while performing symbolic analysis (see §B). This is in fact caused by instances of structs located in the kernel space, preventing the program path from reaching hard-constraint branch points (remaining unexplored). Since HFL is currently interested in fuzzing kernel space via syscall APIs, we are unable to dictate such kernel instances directly. Nonetheless, we expect to handle this as HFL proceeds to see them in continuing code explorations.

Commercial Off-the-shelf Kernels. A few components of HFL (e.g., kernel translation) are deployed during the compilation phase. For this reason, for now, HFL does not support source-free operating systems such as Windows. We believe this could be overcome by adopting a sort of runtime analysis technique (e.g., instrumentation) with extra efforts in the future.

IX. Conclusion

This paper presents HFL, a hybrid fuzzer for testing kernels. We identified three key challenges that undermine the efficiency of both fuzzing and symbolic executor, and design HFL to resolve such challenges. As a result, HFL allows us to bring hybrid fuzzing into kernel space effectively and efficiently. Our evaluation result shows that HFL outperforms each of the approaches, represented by Moonshine, Syzkaller, TriforceAFL, etc, by achieving a higher coverage than them. More importantly, in testing recent Linux kernels with HFL, we found 24 previously unknown vulnerabilities and made/is making them fixed in future releases.

Acknowledgments

We would like to thank the anonymous reviewers for insightful comments which significantly improved the final version of this paper. This work was partly supported by the National Research Foundation (NRF) of Korea grant funded by the Korea government(MSIT) (No. NRF 2019R1C1C1006095).

References

[1] Https://kiwi.cs.purdue.edu/hfl/.

[2] "addr2line," 2018, https://sourceware.org/binutils/docs/binutils/addr2line.html.

[3] "Gcc," 2018, https://gcc.gnu.org.

[4] "Kcov," 2018, https://www.kernel.org/doc/html/v4.15/dev-tools/kcov.html.

[5] "Kernel address sanitizer," 2018, https://github.com/google/kasan/wiki.

[6] "Kernel memory sanitizer," 2018, https://github.com/google/kmsan.

[7] "Undefined behavior sanitizer," 2018, https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[8] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, Hyderabad, India, May–Jun. 2014.

[9] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, p. 50, 2018.

[10] R. Baldoni, E. Coppa, D. C. DâĂŹelia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[11] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008.

[12] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler, "Exe: A system for automatically generating inputs of death using symbolic execution," in *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Oct.–Nov. 2006.

[13] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[14] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[15] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems," *Acm Sigplan Notices*, vol. 46, no. 3, pp. 265–278, 2011.

[16] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: Interface aware fuzzing for kernel drivers," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[17] EPFL, "S2e: The selective symbolic execution platform," 2018, http://s2e.systems/docs/Howtos/Coverage/index.html.

[18] L. Foundation, "llvmlinux," 2017, https://wiki.linuxfoundation.org/llvmlinux.

[19] D. Gens, S. Schmitt, L. Davi, and A.-R. Sadeghi, "K-miner: Uncovering memory corruption in linux," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[20] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, p. 20, 2012.

[21] I. Google, "ClusterFuzz: All Your Bug Are Belong to Us," 2019, https://github.com/google/clusterfuzz.

[22] N. Group, "Triforce linux syscall fuzzer," 2016, https://github.com/nccgroup/ TriforceLinuxSyscallFuzzer.

[23] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations." in *Proceedings of the 22th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2013.

[24] H. Han and S. K. Cha, "Imf: Inferred model-based fuzzer," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[25] Intel, 2016, https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/.

[26] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[27] D. Jones, "Trinity: Linux system call fuzzer," 2011, https://github.com/kernelslacker/trinity.

[28] S. Y. Kim, S. Lee, I. Yun, W. Xu, B. Lee, Y. Yun, and T. Kim, "Cab-fuzz: practical concolic testing techniques for cots operating systems," in *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, Jul. 2017.

[29] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks,

"Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 2123–2138.

[30] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: program-state based binary fuzzing," in *Proceedings of the 16th European Software Engineering Conference (ESEC) / 25th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Paderborn, Germany, Sep. 2017.

[31] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "Dr.checker: A soundy analysis for linux kernel drivers," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1007–1024.

[32] R. Majumdar and K. Sen, "Hybrid concolic testing," in *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, Minneapolis, MN, May 2007.

[33] S. Pailoor, A. Aday, and S. Jana, "Moonshine: Optimizing os fuzzer seed selection with trace distillation," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, USA, Aug. 2018.

[34] B. S. Pak, "Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution," *School of Computer Science Carnegie Mellon University*, 2012.

[35] J. Pan, G. Yan, and X. Fan, "Digtool: A virtualization-based framework for detecting kernel vulnerabilities," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 149–165.

[36] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation." SP18.

[37] D. A. Ramos and D. R. Engler, "Under-constrained symbolic execution: Correctness checking for real code." in *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[38] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb.–Mar. 2017.

[39] Rode0day, "Archived Results: Final Scores for Rode0day-18.10," 2018, https://rode0day.mit.edu/results/4.

[40] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kafl: Hardware-assisted feedback fuzzing for os kernels," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, Aug. 2017.

[41] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 263–272.

[42] Y. Shoshitaishvili, C. Kruegel, G. Vigna, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[43] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz, "Periscope: An effective probing and fuzzing framework for the hardware-os boundary," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[44] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[45] Y. Sui and J. Xue, "Svf: interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th International Conference on Compiler Construction*. ACM, 2016.

[46] D. Vyukov, "Syzkaller," 2015, https://github.com/google/syzkaller.

[47] P. Wang, J. Krinke, K. Lu, G. Li, and S. Dodier-Lazaro, "How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the linux kernel," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 1–16.

[48] W. Wang, K. Lu, and P.-C. Yew, "Check it again: Detecting lacking-recheck bugs in os kernels," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 1899–1913.

[49] V. M. Weaver and D. Jones, "perf fuzzer: Targeted fuzzing of the perf event open () system call," Technical Report UMAINEVMW-TR-PERF-FUZZER, University of Maine, Tech. Rep., 2015.

[50] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim, "Precise and scalable detection of double-fetch bugs in os kernels," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 661–678.

[51] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "Qsym: a practical concolic execution engine tailored for hybrid fuzzing," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, USA, Aug. 2018.

[52] M. Zalewsk, "American fuzzy lop," 2014, http://lcamtuf.coredump.cx/afl.

[53] L. Zhao, Y. Duan, H. Yin, and J. Xuan, "Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[54] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "Firm-afl: High-throughput greybox fuzzing of iot firmware via augmented process emulation," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1099–1114.

APPENDIX

A. Static Dependency Analysis

Table VII presents the result of our static dependency analysis. From the technical perspective, our approach to static points-to analysis is brought from that of Razzer [26]. Specifically, we partition kernel source code according to subsystem classification, and then separately perform the analysis on each partition. Such a partitioning approach not

Category	Analysis Target	Size (.bc)	Analysis Time (h)	# Candidate Pairs
File system	fs/	75 MB	7	110 K
Network	net/	255 MB	90	530 K
Drivers	drivers/	322 MB	83	460 K

TABLE VII: The details of HFL static dependency analysis.

Category	#Prog	#Prog$_S$	#Input$_S$	#Input$'_S$
File system	4.6 M	1,526	521	343
Network	5.1 M	2,121	1,225	632
Drivers	5.0 M	2,034	1,472	851

TABLE VIII: The statistics of hybrid-specific feature in the experiment (§VI-B3). **Prog** and **Prog**$_S$ denote concretely and symbolically executed programs during the experiment, respectively. **Input**$_S$ are new input programs produced as a result of the execution of **Prog**$_S$. Of which, **Input**$'_S$ are the ones actually contributing new execution path.

only allows us to alleviate significant analysis overhead, but fits each partition into each syscall category properly. Considering these factors, we believe this effectively identifies potential dependencies that suit our purpose although it may lead to false negative outcomes due to missing dependencies across partitions. Since such analysis is one time task in our work, we note its non-trivial overhead does not affect the overall performance of HFL.

B. Statistics of Hybrid-Fuzzing

In Table VIII, we illustrate more detailed analysis of the hybrid-specific feature in the coverage experiment (§VI-B3). In the table, **Prog** and **Prog**$_S$ indicate concretely and symbolically executed programs, respectively. **Input**$_S$ in the fourth column represents input programs that are newly generated as a result of symbolic execution. One notable observation is that out of symbolically executed programs (**Prog**$_S$), non-negligible cases do not yield new inputs although they are supposed to trigger tight branches on the particular path. This is mostly caused by non-deterministic states of internal kernel instances which are not under our control. In the fifth and sixth columns, as expected, the new input programs (**Input**$_S$) not only contribute new code paths (i.e., **Input**$'_S$), but also have significant influence on the future input generation, then lead to coverage improvement. This stems from that such new inputs would be pushed into the corpus and reused as a source of mutation, thereby helping trigger other new paths.

Crash type	Description	Kernel
integer overflow	kernel BUG at fs/xfs/xfs_message.c	4.19-rc8
integer overflow	kernel BUG at fs/btrfs/ctree.c	4.17.19
integer overflow	kernel BUG at net/core/skbuff.c	4.19-rc8
task hang	Undefined behaviour in fs/open.c	5.0-rc2
task hang	WARNING in __alloc_pages_slowpath	4.18.20
task hang	task hung in truncate_inode_pages_range	4.20.0
task hang	WARNING in __ext4_handle_dirty_metadata	4.19-rc2
task hang	WARNING in usb_submit_urb	4.19-rc2
task hang	unable to handle kernel paging request in alloc_vmap_area	4.18-rc4
task hang	Undefined behaviour in net/core/sock.c	4.20.2
task hang	task hung in blk_mq_get_tag	4.17.19
task hang	task hung in __fdget_pos	4.17.19
task hang	task hung in __flush_work	5.0-rc2

TABLE IX: List of 13 known vulnerabilities.

Component	Tool	Lines of Code
Fuzzer	syzkaller	840 (Go)
Symbolic Analyzer	s2e-2.0	420 (C++)
Kernel Translator	gcc-7.3	820 (C)
Coordinator	-	480 (Python)

TABLE X: Modifications of the tools used in HFL. Note that *coordinator* acts as a glue in communcation between fuzzer and symbolic analyzer (e.g., user program transformation).