ACTOR: Action-Guided Kernel Fuzzing

Marius Fleischer[1*], Dipanjan Das[1*], Priyanka Bose[1], Weiheng Bai[2], Kangjie Lu[2], Mathias Payer[3], Christopher Kruegel[1], and Giovanni Vigna[1]

[1]University of California, Santa Barbara
[2]University of Minnesota
[3]EPFL
{mariusfleischer,dipanjan,priyanka,chris,vigna}@cs.ucsb.edu
{bai00093,kjlu}@umn.edu
{mathias.payer}@nebelwelt.net

June 16, 2023

Abstract

Fuzzing reliably and efficiently finds bugs in software, including operating system kernels. In general, higher code coverage leads to the discovery of more bugs. This is why most existing kernel fuzzers adopt strategies to generate a series of inputs that attempt to greedily maximize the amount of code that they exercise. However, simply executing code may not be sufficient to reveal bugs that require specific sequences of actions. Synthesizing inputs to trigger such bugs depends on two aspects: (i) the *actions* the executed code takes, and (ii) the order in which those actions are taken. An *action* is a high-level operation, such as a heap allocation, that is performed by the executed code and has a specific semantic meaning.

ACTOR, our action-guided kernel fuzzing framework, deviates from traditional methods. Instead of focusing on code coverage optimization, our approach generates fuzzer programs (inputs) that leverage our understanding of triggered actions and their temporal relationships. Specifically, we first capture actions that potentially operate on shared data structures at different times. Then, we synthesize programs using those *actions* as building blocks, guided by bug templates expressed in our domain-specific language.

We evaluated ACTOR on four different versions of the Linux kernel, including two well-tested and frequently updated long-term (5.4.206, 5.10.131) versions, a stable (5.19), and the latest (6.2-rc5) release. Our evaluation revealed a total of 41 previously unknown bugs, of which 9 have already been fixed. Interestingly, 15 (36.59%) of them were discovered in less than a day.

1 Introduction

The operating system (OS) kernel acts as a mediating layer between the hardware and user-space applications. The kernel's direct, unrestricted access to system resources, particularly physical memory, makes it an appealing target for attackers. Vulnerabilities in the kernel can have disastrous consequences on the entire system. For example, Use-After-Free (UAF) bugs can be exploited to launch a local privilege escalation attack [6], a remote code execution attack [4], or even to break the security boundary by escaping from a container [5].

Popular OS kernels support a wide range of CPU architectures, peripherals, and hardware and software protocols, leading to a large code base. Finding vulnerabilities in a code base of this magnitude is a challenging task. For example, the Linux kernel consists of over 30 million lines of code [2]. Therefore, in recent years, substantial research effort, from both industry and academia, has been directed toward developing new techniques to uncover bugs in OS kernels. Fuzzing, a popular dynamic analysis technique, has shown significant promise toward achieving this goal and has found thousands of bugs in the Linux kernel [17].

The kernel exposes its functionality through system calls (*syscalls*), which are functions that can be invoked from user-space processes, and make for a natural starting point to test the kernel for vulnerabilities. In most approaches, a syscall fuzzer first synthesizes inputs, also called fuzzer *programs*, that consist of a series of syscalls along with their arguments. Once created, the fuzzer executes the programs on the kernel under test and leverages sanitizers [8, 10, 11, 20], in-kernel fault injection [13], runtime verification frameworks [14], and assertions injected by the developers [12] as *oracles* to signal when a bug is triggered.

Since the kernel maintains a vast global state across the invocation of syscalls, it can be seen as a "state machine" with (essentially) infinite states. During their execution, fuzzer-generated programs drive the kernel from one such state to another, looking for latent bugs that may surface only when the kernel is in certain states. Therefore, recent research explored ways to better navigate this state-space by synthesizing effective programs, with *coverage* as the (proxy) metric to measure a fuzzer's success.

If a fuzzer program simply invokes arbitrary syscalls with random arguments, it would inevitably result in poor code and state coverage, as these invocations would fail quickly along shallow error paths. Even if they do not fail, they are likely not to penetrate deep into the kernel code. Thus, to address these challenges, fuzzers adopt different strategies to invoke related

* These authors contributed equally to this work.

syscalls in a meaningful order and with proper arguments. For example, SYZKALLER [18], a popular coverage-guided fuzzer, requires manually written descriptions of inter-syscall relations that are then refined dynamically based on the observed coverage. Other examples are MOONSHINE [45] and HEALER [58], which infer if two syscalls are related, either directly (using static program analysis and dynamic traces) or indirectly (using coverage feedback), with the goal of synthesizing programs that yield higher coverage.

We believe that, given the complex nature of the OS kernel, code coverage alone is insufficient for effective fuzzing. While optimizing for coverage is certainly important, many vulnerabilities are triggered only when the executed code (i) takes certain *actions* (ii) in a specific order. Thus, it is important to augment code coverage with the awareness of (higher-level) actions and their ordering. The former ensures that the fuzzer reaches individual code parts that contain a bug. If a bug is present, then the latter makes it more likely that the bug is actually triggered. For example, to discover a Use-After-Free (UAF) vulnerability, an input is required to reach the *allocation* action (performed by syscall s_a), the *free* action (performed by syscall s_f), and a subsequent *use* action (performed by syscall s_u). In addition, the actions need to operate on the *same* data structure in that *exact order*. A coverage-maximizing fuzzer may execute the three syscalls s_a, s_f, and s_u (triggering their respective actions) as part of different programs, which operate on different data structures. However, unless a fuzzer synthesizes the $s_a \rightarrow s_f \rightarrow s_u$ sequence, the UAF bug will not trigger. This shows that code coverage alone is not enough: A coverage-maximizing fuzzers will execute the right code, but fail to find the bug. This is because it does not execute the code in the right order.

Inspired by the observation above, we present ACTOR, a system that introduces action-awareness to kernel fuzzing. In particular, we propose a novel technique to synthesize potentially bug-inducing programs (inputs) instead of optimizing for code coverage. Our approach follows a two-step process: *action mining*, followed by *program synthesis*.

Action Mining. Unlike traditional coverage-guided fuzzers that view the execution of code as a series of instructions, ACTOR captures it as a sequence of *actions*. Actions are high-level operations with a specific semantic meaning, such as the allocation of memory buffers, the increment of a kernel reference counter, or the writing of a pointer field inside a structure. While a *coverage-guided* strategy strives to generate programs that attain progressively higher coverage, our *action-guided* strategy aims to generate programs that result in actions operating on shared objects. The assumption is that these actions, when executed in the right order, are more likely to trigger bugs.

In our approach, actions are recorded during the execution of a fuzzer program. Interestingly, the notion of coverage-guidance and action-guidance are not conflicting, but complementary. Since triggering diverse actions is challenging because of classic coverage issues, we "piggyback" our action discovery process on a coverage-guided strategy.

Syscalls trigger actions. We first instrument the kernel to observe those actions. Initially, actions might be fairly generic. For example, a heap read could be a read of a value, an array index, or a pointer. Therefore, we refine actions with the help of a static analysis-based step called *semantic labeling*. We then collect the association between a system call and its actions, along with the stack trace of the instruction that triggers the action. We refer to such an association as a *dart*.

We consider two actions to be *related* if they operate on the same memory region. Related darts that come from the same program and operate on a common region are aggregated into a *group*. Finally, multiple smaller groups, potentially coming from different programs, are merged into larger groups on the basis of common stack traces. Following the example above, s_a, s_f, and s_u will end up in the same group G, as they operate on the same region.

Program Synthesis. We design a flexible domain-specific language (DSL) to express and encode a wide range of vulnerability templates, which are used to synthesize likely bug-triggering programs. While we present a diverse set of templates inspired by our observation of real-world bug types, our DSL enables an analyst to easily extend ACTOR by adding support for additional templates, if needed. For example, UAF is encoded as $alloc \rightarrow free \rightarrow [read|write]$ in our DSL.

To synthesize a fuzzer program, ACTOR first selects a group and a bug template. It then chooses appropriate darts from the group and uses syscall information from those darts to instantiate the given template. For example, if the group G is selected, our approach synthesizes the sequence $s_a \rightarrow s_f \rightarrow s_u$, which triggers the UAF bug.

We evaluated ACTOR on four different versions of the Linux kernel, including two well-tested and actively-patched long-term (5.4.206 and 5.10.131) versions, a stable (5.19), and the latest (6.2-rc5) release. In those kernels, ACTOR discovered a total of 41 previously unknown bugs, of which 9 have already been fixed.

In this paper, we make the following contributions:

Action-guided fuzzing. We introduce the notion of *actions* in kernel fuzzing. While coverage-guided fuzzers interpret program execution as a series of instructions, ACTOR views it as a sequence of actions, which is used to drive program generation during fuzzing.

Novel application of program synthesis. While program synthesis has traditionally been used in different research contexts, to our knowledge, we are the first to apply template-guided synthesis for generating fuzzer programs. ACTOR generates programs guided by templates that are written in a domain-specific language, and are more likely to trigger bugs in a vulnerable code snippet.

Prototype implementation and new bugs. We implement our technique in a tool called ACTOR, which discovered 41 previously-unknown bugs in the latest and long-term versions of the Linux kernel. We will release both the code of the tool and the experimental data to facilitate future research [1].

2 Motivation

We first discuss the limitations of existing coverage-guided fuzzers. Then, we motivate the need for *template-guided* program synthesis, on the basis of a real-world example. Specifically, we show that certain sequences of actions are *unlikely* to be inferred by the existing techniques, but they are important to trigger kernel bugs. We also provide an overview of ACTOR, highlighting how it circumvents those limitations.

Order-sensitivity of existing coverage-guided fuzzers. SYZKALLER [18] employs a *choicetable* that records the probability of one syscall getting invoked before another. The choicetable is populated both statically and dynamically. SYZKALLER ships with manually-written system call descriptions that specify their arguments and return types. If a pair of syscalls share arguments of the same type, they are assigned a higher probability. Dynamically, the fuzzer increases the probability for a pair of syscalls when they appear together in a fuzzer program that contributes to new coverage.

A seed distillation system, MOONSHINE [45] distills millions of program traces down to a compact, minimized collection of seeds. Its algorithm is greedy; it favors syscalls that produce the highest coverage. It iterates over the syscalls accumulated from all the traces sorted by descending order of the coverage they produce. If a syscall s yields new coverage, they add s, along with the other syscalls in the program that s depends on to the minimized seed corpus. While MOONSHINE prepares the corpus before fuzzing begins, HEALER [58] performs relation-learning online in the fuzzing loop so that the relations can be continuously refined as the fuzzing progresses. If a fuzzer program $\mathcal{P}$ yields new coverage, HEALER first minimizes $\mathcal{P}$ such that the minimized program $\mathcal{P}'$ exhibits the same coverage as $\mathcal{P}$. Then, it systematically removes each syscall s_i that comes before a syscall s_j, one by one. If the removal of s_i alters the coverage produced by s_j, HEALER learns an influence relation between those two syscalls, which is recorded in a *relation table*.

In summary, state-of-the-art fuzzers learn a *specific* ordering of syscalls to maximize coverage. However, an increase in coverage does not always lead to the discovery of bugs. For example, AGAMOTTO [57] covered 47.8% more paths in drivers, while it found just one new bug. As we will show next, oftentimes a specific sequence of actions is likely to trigger bugs in the code, regardless of the coverage they produce.

Importance of actions. Existing fuzzers are suboptimal due to their bias towards ordering system calls so that they maximize code coverage. Unfortunately, it may happen that multiple different orderings produce near-similar coverage, but only one of those is triggering a bug. How do we choose *that* one out of those many orders? We believe that the answer can be found in the actions that the code performs. Actions provide the necessary signal to better understand the code's behavior. With a thorough analysis of a real-world bug, we will show that, in order to trigger bugs, actions (what the executed code does) are important, in addition to coverage (what code is executed).

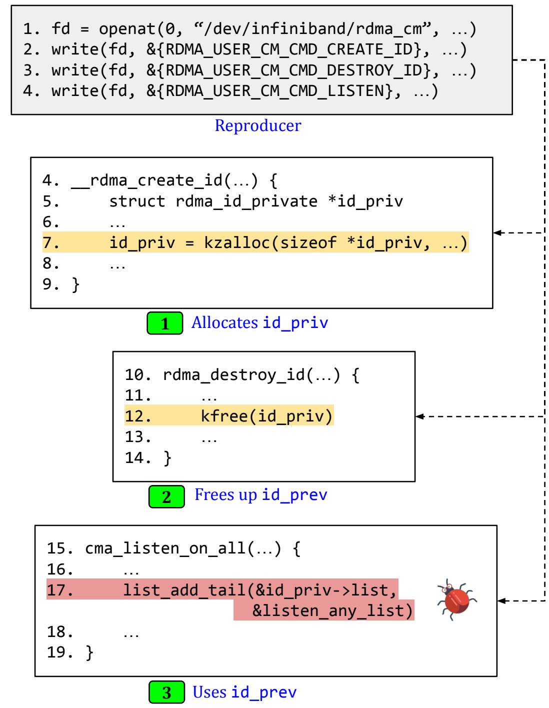

Figure 1: Use-after-free: used `id_priv` after release

> The existing coverage-guided fuzzers are both *action-agnostic* and *order-senstitive*, *i.e.*, not only do they disregard operations that a syscall performs, but also they are likely to favor coverage-maximizing ordering of syscalls over others.

We hope to motivate the importance of actions and order awareness by walking the reader through a real-world bug in the Linux kernel. First, we provide a *reproducer*, which is a fuzzer program that triggers the bug. We then discuss the *actions* taken by those syscalls to trigger the bug.

▶ **Use-After-Free (UAF).** Figure 1 shows a simplified example of a UAF in the Remote Direct Memory Access (RDMA) functionality of the InfiniBand driver. For the sake of presentation, we use the following notation: `{val}` represents a structure, possibly with many fields, where `val` shows the value of one of those fields. Lines 1–4 present the reproducer to trigger the bug. In Line 1, the `openat` syscall acquires a file descriptor `fd` by opening the appropriate device.

Alloc. In Line 2, the second argument of the `write` syscall accepts a pointer to a `rdma_ucm_cmd_hdr` structure (not shown in the figure). The structure has a command code `cmd` field, which is set to `RDMA_USER_CM_CMD_CREATE_ID`. This command invokes 1 `__rdma_create_id()`, which allocates `id_priv` (Line 7), a struct of type `rdma_id_private` (Line 5).

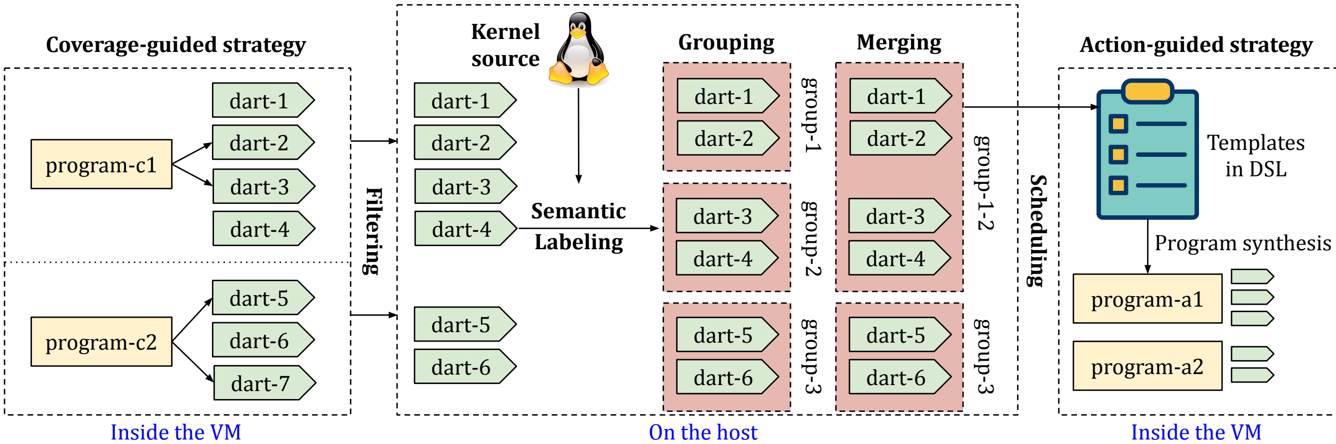

Figure 2: The ACTOR workflow

Free. In Line 3, the `RDMA_USER_CM_CMD_DESTROY_ID` command invokes 2 `rdma_destroy_id()`, which now deallocates the same `id_priv` at Line 12.

Use. In Line 4, the `RDMA_USER_CM_CMD_LISTEN` command invokes 3 `cma_listen_on_all()`, which attempts to put the already-freed `id_priv` in a list. When the `id_priv->list` field is accessed in Line 17, it triggers a UAF bug, because it has already been freed.

Bug-triggering action sequence. In order to trigger this bug, one needs to identify the actions (*action mining*), and then invoke an *alloc* → *free* → *use* action sequence (*program synthesis*), all operating on the same `id_priv` buffer.

Existing coverage-maximizing fuzzers are likely to do the following. From the syscall grammar, they derive that `write` accepts the file descriptor `fd` produced by `openat`, and therefore will infer that an `openat` → `write` sequence is promising to achieve better coverage. Beyond that, they would not specifically attempt to trigger a UAF bug (lack of order-awareness). In fact, they are oblivious to which specific `write` triggers which of the actions required to trigger the bug (lack of action-awareness). Therefore, they would just randomly order those syscalls, and monitor for an increase in coverage.

From analyzing real-world bugs, we learn that: **(i)** often, bugs are only triggered when certain actions occur in a specific order, **(ii)** existing fuzzers are optimized to increase coverage, not necessarily to trigger bugs, and **(iii)** with current fuzzers, a user has no good way to ensure that they are generating programs that conform to a specific structure.

Based on these observations, we design ACTOR, which **(i)** identifies relationships between a syscall and the actions it triggers (action-awareness), **(ii)** uses those actions in a specific order to generate potentially bug-inducing programs (order-awareness), and **(iii)** allows the writing of program specifications in a domain-specific language with precise control over *actions* and their *order*.

3 ACTOR Design

ACTOR, our kernel fuzzing framework, introduces a novel approach to fuzzing, called *action-guided* fuzzing. Complementary to the popular *coverage-guided* program generation strategy (used in most fuzzers) that greedily optimizes for code coverage, our strategy leverages the generated coverage to synthesize potentially bug-inducing programs. Specifically, we use a combination of dynamic analysis and template-based synthesis, where templates are written in a domain-specific language (DSL). Figure 2 shows ACTOR's main components and how they operate along two phases. During *action mining* (Section 3.1), we collect relevant *actions* performed by the executed code, as well as the associated syscalls (along with their arguments) that trigger those actions. We call this information *darts*. In the following *program synthesis* phase (Section 3.2), we stitch those darts together to generate potentially bug-inducing programs.

3.1 Action mining

The goal of the *action mining* phase is to infer relationships between syscalls and the actions that they trigger during execution. The outputs of this phase are action *groups*, which are consumed by the *program synthesis* algorithm. We will first define *actions*, and then explain the stages of the *action mining* phase in more detail.

Actions. Actions are the building blocks of ACTOR's action-guided fuzzing approach. Traditional grey-box fuzzers rely on code coverage as the primary feedback to decide which inputs deserve further exploration. They view program execution (traces) as a stream of low-level instructions, but they are oblivious to the semantics of the executed code. Instead, ACTOR interprets an execution trace as a series of high-level operations, called *actions*, which forms the unit of abstraction that helps our fuzzer to build a deeper semantic understanding.

In principle, one can define a wide range of actions to accommodate multiple classes of bugs. In this work, we use the following types of actions: (kernel) heap allocation ($\mathcal{A}_a$), heap deallocation ($\mathcal{A}_d$), heap value read ($\mathcal{A}_{vr}$), heap pointer read ($\mathcal{A}_{pr}$), heap index read ($\mathcal{A}_{ir}$), heap value write ($\mathcal{A}_{vw}$), heap pointer write ($\mathcal{A}_{pw}$), and heap index write ($\mathcal{A}_{iw}$). $\mathcal{A}_{pr}$ and $\mathcal{A}_{pw}$ refer to the read/write of the value (address) of a pointer. The set of all action types is called the *actiontype set* $\mathbb{AT}$.

We define *action* $a = \langle at, addr, size\rangle$ as a triplet, where $at \in \mathbb{AT}$, *addr* is the address associated with the action, and *size* refers to the size (in bytes) of the memory object that this action operates on. The interpretation of *addr* depends on the action in question. For example, for $\mathcal{A}_a$, it is the base address of the allocated memory region.

A fuzzer program $\mathcal{P} = \{s_1, s_2, \ldots, s_n\}$ consists of a sequence of syscalls, along with their arguments. When the fuzzer runs this (user-mode) program – by invoking each syscall s_i in the sequence – a portion of the kernel code is executed. An *action* is a certain high-level operation that is performed by this executed code and that has a specific semantic meaning associated with it. For example, consider the `kmalloc` function, which allocates a chunk of memory on the kernel heap. While a coverage-guided approach is sensitive to the precise set of instructions executed inside the allocator, our action-guided approach views the invocation of `kmalloc` as one single *allocation* action. Each syscall s_i yields a series of actions $\{a_1, a_2, \ldots, a_n\}$.

As we will discuss in Section 3.2, these actions are sufficient to generate programs targeting a broad class of diverse bugs. We will discuss in Section 6 how the framework can be extended to support additional action types for other bug classes.

Action guidance is, by design, less *granular* than coverage guidance. Continuing with the previous example, allocators contain complex logic to handle diverse heap states and allocation sizes. Depending on the kernel state when the allocator is invoked, paths exercised inside the allocator will be different, and so will the coverage. Now, as explained in Section 2, the core idea behind ACTOR is to synthesize programs that exercise certain actions in a specific order. In our approach, it is important to focus on *which* action is taken (allocation), rather than *how* it is taken (*e.g.*, which specific `slab` cache the allocation happens from, which locks are taken, and so on).

Dart. A *dart* associates a syscall invocation (a syscall along with its arguments) with an action that it triggers. Formally, a *dart* $d = \langle s, a, \Delta\rangle$ is a triplet consisting of a syscall s (along with its arguments) that triggers an action a with a stack trace Δ. If a syscall triggers multiple actions, *e.g.*, a heap allocation followed by a read from the allocated buffer, it will produce one dart for each action. Darts are essential for our fuzzer to be able to re-trigger observed actions. Specifically, during *program synthesis*, ACTOR re-uses the syscall s (with the same arguments) from a dart d, assuming that it triggers the associated action a with the same stack trace Δ.

Note that when a dart is re-executed, the kernel state is often different from when this dart was recorded. Since a state difference can divert the control flow within the syscall, it can interfere with the ability of the dart to trigger the intended action. Fortunately, as we will empirically demonstrate in Section 5, darts have an acceptable success rate of re-execution so that they can effectively be used for program synthesis.

On execution of a program $\mathcal{P} = \{s_1, s_2, \ldots, s_n\}$, we record the actions triggered by the respective syscalls. A syscall s_i triggers actions $\{a_{ij}\}$, where a_{ij} represents the j-th action in the series. We then transform the action a_{ij} to its corresponding dart d_{ij}. The *dart set* $\mathbb{D} = \{d_{ij} | i \in [1, n], j \geq 1\}$ of $\mathcal{P}$ is the set of all darts generated by $\mathcal{P}$. For a dart $d \in \mathbb{D}$, we define the following operators—**(i)** `ts`(d): returns the timestamp when a dart d is generated. **(ii)** `ActsOn`(d): returns the heap allocation that the dart d operates on. **(iii)** `Alloc`(d): returns the heap allocation performed by the dart d, if any. The *allocation set* $\mathbb{A}_t = \{\texttt{Alloc}(d) | d \in \mathbb{D}, \texttt{ts}(d) < t\}$ at time t is the union of all allocations performed by $\mathcal{P}$ until time t. To determine the heap allocation that the dart d operates on, `ActsOn`(d) compares if $d.a.addr$ falls in the range of address $[d_a.a.addr, d_a.a.addr + d_a.a.size]$, where $d_a \in \mathbb{A}_t$.

Dart reduction. Depending on the number and types of actions, a program can generate a large number of darts. For example, it is quite common for a syscall to repeatedly read values from the heap memory, and, therefore, heap value read ($\mathcal{A}_{vr}$) darts are typically frequent. This poses a problem for two reasons: **(i)** the communication overhead to transfer the darts from the guest VM to the host increases proportionally with the number of darts, and **(ii)** as the number of darts increases, it becomes harder for the synthesis algorithm to choose the most effective ones. In other words, redundant darts degrade the performance of the fuzzing loop without yielding any additional benefit.

We observe that many common bugs manifest only when related syscalls interact with each other. The relationship is frequently established through shared memory accesses [45, 58]. The goal of the dart reduction phase is to limit the number of darts without hurting the performance of the fuzzer. To keep the downstream analysis tractable, we enforce two policies based on shared memory access: **(i)** ACTOR keeps darts that operate on previously-observed heap allocations. Consider a dart set $\mathbb{D}$, and the allocation set $\mathbb{A}_t$. We keep a dart $d \in \mathbb{D}$ only if `ActsOn`$(d) \in \mathbb{A}_t$, where $t = \texttt{ts}(d)$. By only retaining darts that access previously-seen allocations, ACTOR not only cuts down the overall number of darts but also ensures that it deals only with related darts going forward. **(ii)** ACTOR records only the first read/write dart per syscall, per allocation. In other words, if a syscall generates multiple read/write darts that operate on the same allocation, we record only the first access for both types.

Dart labeling. Initially, the darts lack high-level semantic information. For instance, when we record a heap read/write action, we do not have any knowledge of what the action semantically means, *i.e.*, if it is a pointer, index, or value read/write. Instead, we first record a *generic* read ($\mathcal{A}_r$)/write ($\mathcal{A}_w$), and then, in this phase, we attempt to refine darts and their action types with the information collected using static source code analysis. Our approach works in two phases: In

Algorithm 1: Semantic labeling of actions

```
1  Function RecoverSemantics
       Input  : Kernel source 𝒦
       Output : Index access types Γ, Pointer access types Ψ
2      Π ← {}, Γ ← {}, Ψ ← {}
3      foreach instruction I ∈ GetAllInstructions(K) do
4          if IsArrayAccess(I) then
5              foreach OP ∈ GetOperands(I) do
6                  RecordStructField(OP)

7      foreach instruction I ∈ GetAllInstructions(K) do
8          if IsStructAccess(I) then
9              Γ ← Γ ∪ IdentifyIndexAccess(I, Π)

10     foreach instruction I ∈ GetAllInstructions(K) do
11         Ψ ← Ψ ∪ IdentifyPointerAccess(I)
12     return Γ, Ψ

13 Function RecordStructField
       Input  : Instruction I
       Output : None
14     if IsStructAccess(I) then
15         struct, field ← GetStruct(I), GetField(I)
           Π[struct] ← Π[struct] ∪ field
16         return
17     if IsLocalVariable(I) then
18         foreach MD ∈ GetMemoryDependencies(I) do
19             RecordStructField(MD)

20     foreach IOP ∈ GetOperands(I) do
21         RecordStructField(IOP)

22 Function IdentifyIndexAccess
       Input  : Instruction I, Struct fields used as indices Π
       Output : Index access performed by I and its type γ
23     γ ← {}
24     struct, field ← GetStruct(I), GetField(I)
25     if field ∈ Π[struct] then
26         if IsLoadInstruction(I) then
27             accessType ← IDX_READ
28         else if IsStoreInstruction(I) then
29             accessType ← IDX_WRITE
30         else
31             return γ
32         γ[I] ← accessType
33     return γ

34 Function IdentifyPointerAccess
       Input  : Instruction I
       Output : Pointer access performed by I and its type ϕ
35     ϕ ← {}
36     if IsPointerValueAccess (I) then
37         if MayReadFromMemory(I) then
38             accessType ← PTR_READ
39         else if MayWriteToMemory(I) then
40             accessType ← PTR_WRITE
41         else
42             return ϕ
43     ϕ[I] ← accessType
44     return ϕ
```

the *semantic recovery* phase, we glean information from the source code; then, we inject this information in the fuzzing loop during the *semantic refinement* phase.

► *Semantic recovery.* The algorithm to determine the access types (index/pointer, read/write) is depicted in Algorithm 1. The top-level `RecoverSemantics` routine iterates over source instructions and dispatches calls to appropriate subroutines. It builds an *index map* Γ and a *pointer map* Ψ that map a source-level read/write instruction to its respective type, *i.e.*, index/pointer read/write. This is a one-time analysis that can be reused over multiple fuzzing runs for a particular kernel build.

Index access. To identify an index read/write, we leverage our observation that when a heap value is used as an index, then typically a structure `S` is allocated on the heap, and one of its fields `S.f` is used as the index. We expect a *structure*, because it would be unusual to allocate a primitive type on the heap.

First, we build a map Π that records the fields of a structure `S` that *influence* array indices, *i.e.*, we record a field `S.f` if it is used in computations that eventually end up in an index (Step I). Next, we use Π to identify any instruction `I` where such a field `S.f` is used. We determine the access type of `I` based on the operation it performs (Step II).

Step I (`RecordStructField`)**.** For an array access operation `arr[i]`, `S.f` can influence `i` in two ways: **(i)** `i` is directly loaded from `S.f` (Line 14), *i.e.*, `i = S.f`, and, in this cases, we record `S.f` in the map Π; or **(ii)** `i` is a local variable (Line 17) loaded indirectly from `S.f` through intermediate variable(s) `v`, *i.e.*, `v = S.f; i = v; arr[i]`. Then, we perform a backward data-flow analysis of `i` to find out `v`, and then recursively call `RecordStructField` on `v` (Line 19). In the end, Π contains all the structure fields `S.f` that influence array indices.

Step II (`IdentifyIndexAccess`)**.** In this phase, for each instruction I that accesses a structure field `S.f`, we consult Π to check (Line 25) if `S.f` has been used as an index. If yes, then we label `I` as $\mathcal{A}_{ir}$ or $\mathcal{A}_{iw}$ depending on whether it is a *load* or a *store* instruction, respectively. In the end, we output Γ, the list of all index reads/writes.

This two-step approach is necessary. Consider this example: $\mathtt{I}_1$`: S.f++;` $\mathtt{I}_2$`: i = S.f;` $\mathtt{I}_3$`: arr[i].` If the analysis encounters $\mathtt{I}_1$ first, it will not know that it is an index write (and not a value write), unless it has seen $\mathtt{I}_2$ and $\mathtt{I}_3$ already. Since there is no guarantee that the analysis will always encounter instructions in the required order, we had to resort to this two-phase approach. Therefore, we first collect the interesting structure fields (`S.f`), and then label all instructions ($\mathtt{I}_1$ as $\mathcal{A}_{iw}$, and $\mathtt{I}_2$ as $\mathcal{A}_{ir}$) that use those fields.

Pointer access. To identify a pointer read/write, we check if an instruction accesses a pointer value (Line 36). If it is, then we label it $\mathcal{A}_{pr}$ or $\mathcal{A}_{pw}$ depending on whether it reads from (Line 37), or writes to (Line 39), the memory. The analysis builds Ψ, the list of all pointer reads/writes.

► *Semantic refinement.* We leverage the information derived from the previous phase to determine the true action types of the darts. For a dart $d = \langle s, a, \Delta \rangle$ of type $\mathcal{A}_r/\mathcal{A}_w$, we leverage the debug information compiled into the kernel image to recover the source instruction `I` from the stack trace Δ. We use `I` to look up both Γ and Ψ to determine if a is an index/pointer read/write. If `I` is found in any of those maps, we label the dart accordingly. Otherwise, we assume that the dart performs a read/write of a value, and its type is changed to $\mathcal{A}_{vr}/\mathcal{A}_{vw}$, respectively.

Dart grouping. From a dart set $\mathbb{D}$, ACTOR forms dart

Bug class	Template	Bug class	Template
Use After Free	$\mathcal{A}_a \to \mathcal{A}_d \to [\mathcal{A}_r \vert \mathcal{A}_w]$	Null Ptr Deref	$\mathcal{A}_a^x \to \mathcal{A}_d^x$
Double Free	$\mathcal{A}_a \to \mathcal{A}_d \to \mathcal{A}_d$	Invalid Free	$\mathcal{A}_d$
Out of Bounds (1)	$\mathcal{A}_a \to \mathcal{A}_{iw}^* \to \mathcal{A}_{ir}$	Memory Leak (1)	$\mathcal{A}_a^*$
Out of Bounds (2)	$\mathcal{A}_a \to \mathcal{A}_{pw}^* \to \mathcal{A}_{pr}$	Memory Leak (2)	$\mathcal{A}_a \to \mathcal{A}_{pw} \to \mathcal{A}_d$
Uninitialized Read	$\mathcal{A}_a \to \mathcal{A}_r$	-	-

Table 1: The bug templates defined by ACTOR

groups to combine related darts. Each group g contains one allocation dart $d_a \in \mathbb{D}$, and all other darts that operate on the allocation `Alloc`(d_a) performed by d_a, *i.e.*, the group $g = \{d | \texttt{ActsOn}(d) = \texttt{Alloc}(d_a)\}$.

Group merging. Grouping works on a dart set, which includes all darts generated by running a single program. To expand our scope and also attempt to learn relationships between syscalls invoked by *different* programs, we perform a *merging* step. This step works on groups that are generated from multiple programs. Consider two groups g_1 and g_2. If the stack trace and action type of a dart $d_1 \in g_1$ in the first group match those of a dart $d_2 \in g_2$ in the second group, then ACTOR merges the darts from both the groups to form a larger group $g_{12} = g_1 \cup g_2$, and discards the parent groups.

The intuition behind merging builds on the *transitivity* of the relation between the darts. Suppose that *grouping* initially generates two groups g_1 and g_2. These two groups contain darts that operate on the same allocation (by definition). In other words, all the darts in a group are related to other darts within the same group. Merging discovers that darts $d_1 \in g_1$ and $d_2 \in g_2$ have identical stack traces. Since the darts are generated from the same program context (stack trace), they are semantically the same. This *semantic similarity* represents our notion of relation between syscalls. Thus, d_1 and d_2 are related. Since we know that both d_1 and d_2 are related to all other darts in their respective groups, we conclude that all the darts in both groups are related, too. Hence, the larger groups produced by our *merging* strategy contain related syscalls.

3.2 Program synthesis

In *program synthesis*, we consume the groups produced by the *action mining* phase. We specify bug templates for real-world bugs using a domain-specific language (DSL). We instantiate those templates by choosing appropriate darts from the groups to generate potentially bug-inducing programs.

Supported bug templates. We have defined nine bug templates (listed in Table 1) to capture important classes of bugs. As we will show later, it is easy to expand this list with additional templates in the future. Note that though the templates increase the probability of triggering targeted bug types, the generated programs can still trigger other bugs as well. For the sake of our presentation, instead of using DSL, we use a form of regex-like expression to describe relevant sequences of actions. In reality, a regex is not as expressive as our DSL is. With regex, it is not possible to express relations like two darts have to be the same, or same/different darts are to be chosen to repeat an action, *etc.*

We use $\mathcal{A}_r := \mathcal{A}_{vr} | \mathcal{A}_{pr} | \mathcal{A}_{ir}$ for a generic read, and $\mathcal{A}_w := \mathcal{A}_{vw} | \mathcal{A}_{pw} | \mathcal{A}_{iw}$ for a write action. We are presenting the bug templates in Table 1.

▶ Use After Free (UAF): We first allocate a buffer, deallocate it, and then attempt to access (read/write) the allocation.

▶ Double Free (DF): We first allocate a buffer, and then perform two deallocations, hoping that the second one would trigger the bug.

▶ Out of Bounds (OOB-1): Consider a structure `S` with two fields: `S.i` (integer) and `S.arr` (array). Moreover, `S.i` is used to index into `S.arr`. Indices are often incremented inside loops. We expect that if we repeatedly use a $\mathcal{A}_{iw}^*$ dart, it might increment `S.i` beyond the length of `S.arr`, so that the next access $\mathcal{A}_{ir}$ would trigger an OOB bug.

▶ Out of Bounds (OOB-2): Consider a structure `S` with a pointer field `S.p` that points to an object `O`. Repeated writes $\mathcal{A}_{pw}^*$ of `S.p` could increment the pointer past the end of `O`. Next, a pointer read $\mathcal{A}_{pr}$ would dereference the pointer, and trigger an OOB.

▶ Uninitialized Read (UR): We force a read $\mathcal{A}_r$ immediately after the allocation $\mathcal{A}_a$ to read from uninitialized memory.

▶ Null Pointer Dereference (NPD): Oftentimes, arrays in the kernel code hold pointers to allocated memory objects, and a separate counter `c` records the number of such objects. In presence of a bug, `c` is incremented first, even if the allocation of an object `O` fails. Imagine, we perform n number of $\mathcal{A}_a$, one of which fails, setting a pointer p to `NULL`. Also, c incorrectly gets set to n, which makes it possible to force n number of $\mathcal{A}_d$. The $\mathcal{A}_d$ on p would trigger an NPD.

▶ Invalid Free (IF): One single $\mathcal{A}_d$ dart could trigger an IF if the syscall forgets to check the validity of the pointer, which is supposed to point to a valid memory object, before invoking the deallocator.

▶ Memory Leak (ML-1): Pointers to allocated memory objects may be stored in a buffer of fixed size, *e.g.*, a ring buffer. Enough allocations $\mathcal{A}_a^*$ may overflow the buffer, thus accidentally overwriting pointers to previously allocated objects. Buffers with lost references can no longer be freed, thus causing a memory leak.

▶ Memory Leak (ML-2): Suppose there are two memory objects O_1 and O_2. O_2 is referenced from a pointer field O_1`.p`. If we free O_1 without freeing O_2 first, the kernel loses the reference to O_2, which results in a memory leak. For our bug template, we allocate O_1, write the pointer to O_2 into O_1`.p`, and then deallocate O_1.

Domain-specific language (DSL). The templates presented above, in reality, are specified using our domain-specific language (DSL). Our DSL design is motivated by the following reasons—**(i)** We do not claim completeness in terms of the bug templates ACTOR ships with. Therefore, we design a DSL to let an analyst specify additional bug templates. This is optional; ACTOR can be used as-is with the default templates. **(ii)** While theoretically, it could be possible to make modifications to the fuzzing engine directly to incorporate a new template, we wanted to shield the user from understanding the complexity of the implementation internals, thus making it usable. For example, with the DSL we designed, it took us only up to 5 minutes to add support for each bug type we presented.

Each bug template $bt = \{\langle at, id, repeat\rangle\}$ is a list of 3-tuples, where $at \in \mathbb{AT}$ is the desired action type, an *id* (specified below), and a repetition specifier *repeat* that specifies the number of times the respective dart needs to be repeated. The program synthesis engine in ACTOR consumes a bug template and substitutes each tuple with one or more darts (determined by the *repeat* field) chosen from the selected groups.

id is an arbitrary non-zero integer that serves two purposes: **(i)** a positive *id* creates a connection between two tuples in a bug template. Specifically, if two tuples have the same *at* and *id* fields, ACTOR will use the same dart for both. For a valid template, the same *id* implies the same *at* as well. Also, a positive *id* makes the engine use the same dart for all repetitions. **(ii)** a negative *id* instructs the engine to pick a new dart for each repetition.

repeat is an optional element that can have three possible values—**(i)** if the field is omitted, then only a single dart is used. **(ii)** if $repeat = X$ (X is an integer), then the dart is repeated exactly X times. Whether the same or different dart will be used is determined by whether the *id* field is positive (same) or negative (different). **(iii)** if $repeat = rX$, then rX (*e.g.*, $r3$) is treated as a meta-variable. Two tuples with the same meta-variable will be repeated the same, yet random, number of times, uniformly chosen from $[1,20]$.

The DSL primarily allows one to specify the relative ordering of actions, the number of repetitions, and influence the selection of darts (optionally, with respect to multiple actions). It does not support expressing any control/data-flow primitives. Below we show how our DSL can specify bug templates discussed in the previous section.

Use After Free (UAF). The bug template for UAF is $\mathcal{A}_a \rightarrow \mathcal{A}_d \rightarrow [\mathcal{A}_r|\mathcal{A}_w]$. Since our DSL does not have support for *OR* operator, this template is specified by a set of six rules covering the six types of reads and writes. The only difference between these rules is the action type specified in the last tuple. For example, the rule for a pointer read is specified by the following list: $bt^{pr}_{UAF} = \{\langle\mathcal{A}_a,1\rangle, \langle\mathcal{A}_d,2\rangle, \langle\mathcal{A}_{pr},3\rangle\}$.

Double Free (DF). The bug template for DF is $\mathcal{A}_a \rightarrow \mathcal{A}_d \rightarrow \mathcal{A}_d$. This template can easily be specified by the following list of three tuples: $bt_{DF} = \{\langle\mathcal{A}_a,1\rangle, \langle\mathcal{A}_d,2\rangle, \langle\mathcal{A}_d,3\rangle\}$. Since the bug class does not require the deallocation to be triggered by the same instruction, we chose to not reuse the same dart for both the $\mathcal{A}_d$ actions to allow for more freedom to the synthesis engine. We omit the optional *repeat* field because we do not want any of the darts to be repeated. Otherwise, that could also be explicitly set to 1. Alternatively, we could also specify this bug pattern as $bt_{DF} = \{\langle\mathcal{A}_a,1\rangle, \langle\mathcal{A}_d,-2,2\rangle\}$. Here, we use a negative *id*, combined with $repeat = 2$ to declare that this template requires two independently picked darts.

Out of Bounds (OOB-1). The bug template for OOB-1 is $\mathcal{A}_a \rightarrow \mathcal{A}^*_{iw} \rightarrow \mathcal{A}_{ir}$. This template can be represented as $bt_{OOB(1)} = \{\langle\mathcal{A}_a, 1\rangle, \langle\mathcal{A}_{iw}, 2, r1\rangle, \langle\mathcal{A}_{ir}, 3\rangle\}$. We use the meta-variable $r1$ to ensure that the number of repetitions for the second tuple is randomly picked.

Out of Bounds (OOB-2). The bug template for OOB-2 is equivalent to the template for OOB-1 except for the type of read and write. Therefore, the template can be expressed as $bt_{OOB(2)} = \{\langle\mathcal{A}_a,1\rangle, \langle\mathcal{A}_{pw},2,r1\rangle, \langle\mathcal{A}_{pr},3\rangle\}$.

Uninitialized Read (UR). The bug template for UR is $\mathcal{A}_a \rightarrow \mathcal{A}_r$. This template can simply be specified by the list $bt_{UR} = \{\langle\mathcal{A}_a,1\rangle, \langle\mathcal{A}_r,2\rangle\}$.

Null Pointer Dereference (NPD). The bug template for NPD is $\mathcal{A}^x_a \rightarrow \mathcal{A}^x_d$. Here, we need to specify a meta-variable to ensure that the $\mathcal{A}_a$ and $\mathcal{A}_d$ get repeated the same number of times. This template can be expressed as $bt_{NPD} = \{\langle\mathcal{A}_a,1,r1\rangle, \langle\mathcal{A}_d,2,r1\rangle\}$ with our DSL. $r1$ signals the synthesis engine that the respective tuples are connected, and they have to be repeated a random, yet equal number of times.

Invalid Free (IF). The bug template for IF is $\mathcal{A}_d$. This template can be specified by the following list of 1 element: $bt_{IF} = \{\langle\mathcal{A}_d,1\rangle\}$.

Memory Leak (ML-1). The bug template for ML-1 is $\mathcal{A}^*_a$. This template can be specified using a meta variable by $bt_{ML(1)} = \{\langle\mathcal{A}_a,1,r1\rangle\}$. The meta variable $r1$ ensures that the number of repetitions is randomly picked.

Memory Leak (ML-2). The bug template for ML-2 is $\mathcal{A}_a \rightarrow \mathcal{A}_{pw} \rightarrow \mathcal{A}_d$. The corresponding specification in our DSL is $bt_{ML(2)} = \{\langle\mathcal{A}_a,1\rangle, \langle\mathcal{A}_{pw},2\rangle, \langle\mathcal{A}_d,3\rangle\}$.

Template-guided synthesis. ACTOR's synthesis engine (SE) consumes two inputs: the bug templates and the dart groups. Recall that our action-guided strategy is complementary to coverage-guided exploration, so we run our synthesis algorithm in parallel to a traditional fuzzer. Specifically, when a fuzzer's generation/mutation routine is invoked to generate the next program, we call our synthesis engine with a probability p ($p = 0.5$ in our implementation).

During synthesis, the engine first chooses a bug template bt from the set of available templates (with uniform probability). It also records the types of darts needed to instantiate a program based on that template. Next, it chooses a group g from the set of available groups (again, with uniform probability). If g does not contain all the required types of darts, then a new group is picked. This process is repeated until the synthesis engine finds a group g with all the required types of darts, or a threshold th number of attempts ($th = 400$ in our implementation) is reached (in which case it gives up). If it finds an appropriate group, then the required number and types of darts are chosen from each type, as specified in the template, to produce a new program to be used as the next fuzzer input.

4 Implementation

We implemented the fuzzer component of ACTOR on top of SYZKALLER [18] (commit: `0d5abf15`). For semantic labeling, we developed a static analysis pass on LLVM 14 [19]. To record actions, we developed a Linux kernel module and modified the Kernel Address SANitizer [10] (KASAN). Our kernel module is mostly self-contained (850 LoC), meaning that the changes made to the core part of the kernel are minimal (49 LoC). Therefore, our modifications can be ported across different kernel versions with relative ease.

Recording actions. We instrument KASAN, a dynamic memory error detector for the Linux kernel, to intercept actions of interest (*i.e.*, heap allocations, heap deallocations, heap reads, and heap writes). KASAN uses a shadow memory to keep track of whether each byte of memory is safe to access. To update the shadow memory and to verify memory accesses, KASAN leverages two types of hooks: **(i)** it instruments kernel memory allocator APIs to intercept heap allocation/deallocation, and **(ii)** the compiler inserts `__asan_load*(addr)` and `__asan_store*(addr)` function calls before each memory access of size 1,2,4,8 or 16 bytes to intercept heap reads and writes. We instrument these hooks to call into our kernel module, passing on information such as the address of the access, the allocation size, and the access type (*alloc*/*free*/*read*/*write*).

Our custom kernel module *actrack* records the intercepted actions and exposes the same to the user-space through a `debugfs` file. The module aims to collect actions that are related to syscall inputs. Thus, we do not collect actions in soft/hard interrupts and some inherently non-deterministic parts of the kernel, *e.g.*, scheduler, locking, *etc.* Action collection is enabled on a per-process basis. We store action-tracking metadata by extending the `task_struct`, a structure that holds process-related information and that is instantiated once for every process created. *actrack* provides an appropriate `ioctl` interface to initialize, enable, and disable action tracking for a particular process.

SYZKALLER [18] has three main components: **(i)** `syz-fuzzer`, the fuzzing engine, **(ii)** `syz-executor`, which executes fuzzer-generated programs to test the target kernel, and **(iii)** `syz-manager`, which coordinates multiple fuzzer instances. Components (i) and (ii) run on the guest virtual machine running the target kernel, while component (iii) runs on the host. As we describe below, we modify different SYZKALLER components to implement our fuzzing strategy.

Figure 3 shows how the kernel-space components (*actrack* and KASAN) record actions for each syscall and then propagate those up to the user-space components (*fuzzer* and *executor*). To execute a program $\mathcal{P}$, the *fuzzer* invokes $\boxed{1}$ the *executor*. When the *executor* is spawned $\boxed{2}$, action tracking starts disabled $\boxed{3}$. Then, the *executor* maps $\boxed{4}$ two shared memory regions: *shmem-1* (between *fuzzer* and *executor*), and *shmem-2* (between *executor* and *actrack*). The required size of the shared memory regions and the cost to propagate actions across layers

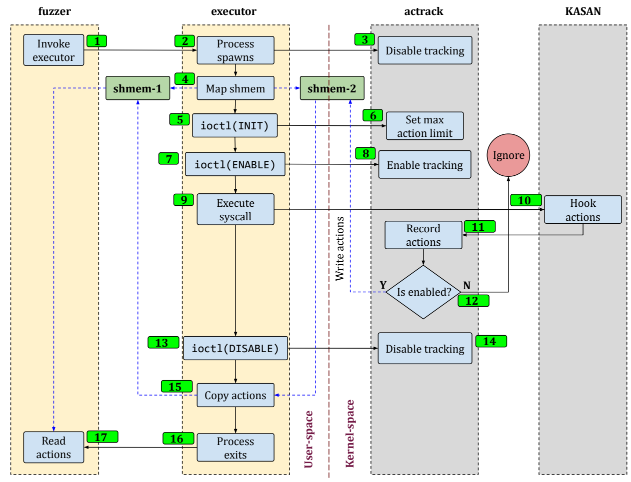

Figure 3: The interaction between the user-space components (*fuzzer* and *executor*), and the kernel-space components (*actrack* and KASAN). Actions are recorded in-kernel for each syscall invoked, and then propagated up to the user-space

grow in proportion to the number of actions recorded. Therefore, the *executor* sets an upper limit $\boxed{6}$ for the number of recorded actions while initializing $\boxed{5}$ *actrack*. Next, the *executor* invokes an `ioctl` $\boxed{7}$ to enable action tracking $\boxed{8}$ for the *executor* process itself, and then immediately calls $\boxed{9}$ the first syscall s of the fuzzer input program $\mathcal{P}$. Actions generated on execution of s are intercepted $\boxed{10}$ by KASAN, which then calls back $\boxed{11}$ into *actrack* to record those actions. *actrack* checks $\boxed{12}$ whether action tracking is enabled, and whether the number of actions recorded so far is within the threshold. If this is the case, then our module writes the action information to *shmem-2*. For every action generated, the callback method in this module is called. To prevent unrelated actions from getting recorded, the *executor* immediately invokes another `ioctl` $\boxed{13}$ to disable action tracking $\boxed{14}$ for itself. The *executor* repeats steps $\boxed{7}$ through $\boxed{14}$ for each syscall in $\mathcal{P}$ (this is not shown in the figure). In the end, it copies $\boxed{15}$ all the actions from *shmem-2* to *shmem-1* to share with the *fuzzer*. Finally, the *fuzzer* reads $\boxed{17}$ the actions from *shmem-1* when the *executor* process exits $\boxed{16}$.

Filtering redundant actions. A program can generate a large number of actions. To remain scalable, we only consider actions with known, matching allocations. In other words, we discard a *read*/*write*/*free* action, unless it operates on a buffer that has been allocated by one of the *alloc* actions that we have seen until that point. Therefore, to decide if an action $\boxed{10}$ is worth recording $\boxed{11}$ (Figure 3), we need to look up the address a associated with the action against a list of allocations. For that reason, we store the allocations key-ed by their addresses in a red-black tree inside the *actrack* module to enable fast lookup.

Sending actions from guest to host. The actions we collect inside the *fuzzer* need to be passed to the *manager*, because that is where the action grouping and merging take place. SYZKALLER uses a Remote Procedure Call-based (RPC) mechanism to send code coverage information from the *fuzzer* to the *manager*. We first attempted to piggyback onto the same communication channel to send our action data. However, the amount and frequency of sending action data is higher than that for the coverage. As a result, RPC turned out to be too slow for our case. Therefore, we use Inter-VM Shared Memory [9] (`ivshmem`) to share a memory region between a QEMU guest and the host. QEMU exposes the memory region as a PCI device to the guest. Shared memory offers higher throughput than RPC, which is leveraged by the *fuzzer* to send the action data to the *manager* running on the host at a faster rate.

5 Evaluation

We find answers to the following research questions in our evaluation: **RQ1.** Can ACTOR find new bugs? **RQ2.** Can darts successfully trigger recorded actions when re-executed on a different kernel state? **RQ3.** Is ACTOR able to trigger more shared accesses than SYZKALLER? **RQ4.** Does ACTOR generate more programs with likely bug-inducing patterns than SYZKALLER? **RQ5.** Can ACTOR learn syscall relations that SYZKALLER does not find? **RQ6.** How does ACTOR compare to the state-of-the-art fuzzers in terms of bugs and coverage?

Experimental setup. We performed our experiments on a server equipped with 2× Intel(R) Xeon(R) E5-2690 v2 @ 3.00GHz CPU and 256 GiB of memory running Ubuntu 20.04.4 LTS 64 bit OS.

To answer **RQ1**, we chose long-term versions of Linux (5.4.206 and 5.10.131), a stable release (5.19), and the latest (6.2-rc5) release. These were the kernel versions for which the Linux kernel maintainers accepted patches for new bugs at the time of our experiments. We fuzz each of these kernels for 12 days. All fuzzer instances used 4 VMs, each having 4 GiB of RAM, and 2 CPU cores. While compiling the kernels, we enabled KCOV to collect code coverage. Since ACTOR relies on KASAN for action mining, we always enabled KASAN when ACTOR was involved. We leveraged KASAN, KMSAN, and KMEMLEAK to detect memory errors in **RQ1**.

RQ2–**RQ6** expose properties of Actor, not the underlying kernel, which makes corresponding evaluations kernel-version agnostic. We chose the kernel used during development (5.17) for these experiments. Each experiment was run for 24 hours and repeated 5 times. We report the average values of the results to limit the impact of randomness, except for **RQ6** where we combined the bugs discovered in all the runs.

To study the effectiveness of ACTOR's program synthesis, we designed SYZKALLER+, which is SYZKALLER with additional logging enabled. This allows us to track actions triggered by vanilla SYZKALLER (and collect the metrics used for this evaluation).

5.1 New bug discovery (RQ1)

To demonstrate the bug-finding abilities of our fuzzer, we run ACTOR on kernel 5.4.206 (LTS), 5.10.131 (LTS), 5.19 (stable), and 6.2-rc5 (latest) for a period of 12 days each. We specifically chose Long-Term Support (LTS) kernel versions because they are maintained by the kernel community over a long time (several years), and bug fixes are regularly back-ported or applied. In addition, SYZBOT [17] fuzzes the Linux kernel continuously with the latest SYZKALLER version, deploying significant resources to do so. Despite this high bar, ACTOR found a total of 41 previously unknown bugs (zero-days). Moreover, 15 (36.59%) of them were discovered in less than a day.

We only reported a bug to the kernel developers if we could generate a *reproducer* for it. We managed to do so for 24 bugs. For 17 bugs, reproducer generation failed due to the well-known statefulness issue of the kernel. Until the time of writing, the developers confirmed 13 bugs, and already patched 9. The details of the bugs we discovered are presented in Table 2.

Another bug, the UAF in `reiserfs_fill_super`, was first discovered by SYZKALLER in 2020, and got fixed shortly after. Again, SYZKALLER discovered the same bug in 2022 in the `linux-next` kernel tree, but it got auto-closed due to SYZKALLER not being able to find a reproducer. ACTOR could not only trigger the bug in 5.19, but also we could generate a reproducer. We already reported the bug to the developers. ACTOR clearly demonstrates its ability to discover bugs that are hard to discover by the state-of-the-art kernel fuzzers.

Manual analysis of the bugs we discovered shows that 63.41% are memory corruption bugs, largely discovered through KASAN and KMSAN. 26.83% of the bugs are discovered through WARNINGs, INFOs and assertions, suggesting that the underlying root causes of these bugs are related to logical errors. The remaining bugs are page faults and other kinds of memory-related bugs.

RQ1: ACTOR found 41 previously unknown bugs in four LTS, stable, and latest versions of the Linux kernel.

Case Study: One of the bugs discovered by ACTOR, the warning in `inet_sock_destruct`, was initially reported by SYZKALLER in 2017, but considered as *fixed* in 2018 because SYZKALLER was not able to trigger it anymore. However, ACTOR was not only able to re-trigger this bug in kernel 5.19, but we were also able to provide the developers with a reproducer, which led to this bug getting patched.

The root cause of this bug is a race condition between closing a `socket` (deallocation of a structure) and transmitting/re-transmitting data buffered by that `socket` (read/write accesses to the `socket` structure). ACTOR's action-guided technique helps to discover this bug as the kernel needs to perform concurrent actions on the same `socket` object. ACTOR synthesized programs that performed specified actions on the same `socket`, which is why we were able to trigger this bug, while other general-purpose fuzzers could not.

Kernel crash description	Crash type	Kernel version	Repro	Reported	Patched	Days
KASAN: use-after-free Read in drm_gem_object_release	Use-After-Free	v5.4.206	✓	✓	✓	0.73
UBSAN: undefined-behaviour in tdp_page_fault	Undef Behavior	v5.4.206	✗	-	-	0.73
general protection fault in sock_def_error_report	GPF	v5.4.206	✓	✓	-	2.96
general protection fault in dd_insert_requests	GPF	v5.4.206	✗	-	-	3.77
general protection fault in reset_interrupt	GPF	v5.4.206	✗	-	-	4.74
BUG: unable to handle kernel paging request in imageblit	Page Fault	v5.4.206	✓	✓	-	6.55
KASAN: use-after-free Read in tcp_retransmit_timer	Use-After-Free	v5.4.206	✗	-	-	6.45
general protection fault in vmx_vmenter	GPF	v5.4.206	✗	-	-	11.30
WARNING: ODEBUG bug in netdev_run_todo	Reachable Warning	v5.4.206	✗	-	-	8.54
UBSAN: undefined-behaviour in xprt_calc_majortimeo	Undef Behavior	v5.4.206	✗	-	-	11.40
KASAN: vmalloc-out-of-bounds Write in snd_pcm_hw_params	OOB access	v5.10.131	✓	✓	✓	0.03
KASAN: null-ptr-deref Write in rhashtable_free_and_destroy	Null ptr deref	v5.10.131	✗	-	-	1.01
WARNING: kmalloc bug in kvm_arch_prepare_memory_region	Reachable Warning	v5.10.131	✗	-	-	5.02
KASAN: use-after-free Read in post_one_notification	Use-After-Free	v5.10.131	✓	✓	✓	1.63
KASAN: vmalloc-out-of-bounds Write in imageblit	OOB access	v5.10.131	✗	-	-	6.49
INFO: task hung in gfs2_read_super	Reachable Info	v5.10.131	✓	✓	-	3.83
general protection fault in start_motor	GPF	v5.10.131	✓	✓	-	0.61
BUG: soft lockup in br_multicast_port_group_expired	Soft Lockup	v5.10.131	✗	-	-	4.45
KASAN: slab-out-of-bounds Read in ntfs_get_ea	OOB access	v5.19-rc6	✓	✓	✓	4.36
KASAN: vmalloc-out-of-bounds Read in cleanup_bitmap_list	OOB access	v5.19-rc6	✓	✓	-	4.11
KASAN: use-after-free Read in run_unpack	Use-After-Free	v5.19-rc6	✓	✓	✓	4.35
KASAN: use-after-free Read in __io_remove_buffers	Use-After-Free	v5.19-rc6	✓	✓	✓	4.78
KASAN: invalid-free in __io_uring_register	Invalid Free	v5.19-rc6	✓	✓	✓	7.29
KASAN: use-after-free Read in reiserfs_fill_super	Use-After-Free	v5.19-rc6	✓	✓		9.53
INFO: task hung in __bread_gfp	Reachable Info	v5.19-rc6	✓	✓	-	2.33
WARNING in inet_sock_destruct	Reachable Warning	v5.19-rc6	✓	✓	✓	6.57
kernel BUG in f2fs_new_node_page	Reachable Assertion	v5.19-rc6	✓	✓	✓	9.10
kernel BUG in ntfs_read_folio	Reachable Assertion	v5.19-rc6	✓	✓	-	0.13
INFO: task hung in __floppy_read_block_0	Reachable Info	v5.19-rc6	✓	✓	-	1.94
BUG: unable to handle kernel paging request in kvm_dev_ioctl	Page Fault	v6.2-rc5	✗	-	-	0.17
KMSAN: uninit-value in __dma_map_sg_attrs	Uninit Value	v6.2-rc5	✓	✓	-	0.94
KMSAN: uninit-value in sr_check_events	Uninit Value	v6.2-rc5	✓	✓	-	0.05
KMSAN: uninit-value in post_read_mst_fixup	Uninit Value	v6.2-rc5	✓	✓	-	0.25
general protection fault in get_cpu_entry_area	GPF	v6.2-rc5	✗	-	-	0.02
KMSAN: uninit-value in nilfs_add_checksums_on_logs	Uninit Value	v6.2-rc5	✓	✓	-	0.45
KMSAN: uninit-value in generic_bin_search	Uninit Value	v6.2-rc5	✓	✓	-	0.01
BUG: unable to handle kernel NULL pointer dereference in ntfs_iget5	Null ptr deref	v6.2-rc5	✓	✓	-	0.12
BUG: unable to handle kernel paging request in get_cpu_entry_area	Page Fault	v6.2-rc5	✗	-	-	0.02
BUG: unable to handle kernel paging request in bpf_ringbuf_alloc	Page Fault	v6.2-rc5	✗	-	-	2.47
KASAN: null-ptr-deref Read in gfs2_evict_inode	Null ptr deref	v6.2-rc5	✗	-	-	0.45
KASAN: null-ptr-deref Read in soft_cursor	Null ptr deref	v6.2-rc5	✗	-	-	2.33

Table 2: New bugs found by ACTOR. ✓: reproducer generated/ reported/ patched, ✗: the crash logs were not sufficient to extract reproducers, ◕: reproducer extraction is still ongoing.

5.2 Re-execution success (RQ2)

When ACTOR synthesizes a new program, it instantiates a bug template with darts that were previously recorded. Of course, these darts will execute on a kernel state that is likely different from the one on which they were recorded. These differences might prevent a dart's ability to re-trigger the underlying action. In this experiment, we measure the fraction of darts that are able to re-trigger the same action, as successful re-execution is important for our approach to work effectively.

Recall that our synthesis engine chooses a bug template bt and a group g when generating a program $\mathcal{P}$. Assume that a dart $d = \langle s,a,\Delta \rangle$ from g is used in $\mathcal{P}$. The dart, when re-executed, generates actions $R_a = \{a_i, \Delta_i\}$. We declare *success* if any one of the triggered actions and its stack trace match the previously recorded one, *i.e.*, $(a,\Delta) \in R_a$. If bt starts with allocation(s) ($\mathcal{A}_a^*$), and one of the allocation(s) fail(s), we exclude subsequent syscalls (generated by bt from g) from our check. The reason is that when an allocation fails, we cannot expect subsequent accesses to this object to succeed.

During our experiments, we noticed that 14 out of a total of 2,072 syscalls exhibited poor re-execution success. That is, they were almost never able to re-trigger recorded actions. Upon further investigation, we realized that the operations they perform are not repeatable unless the kernel state is *reset*. For example, the `mount` syscall mounts a new file system. Re-executing this dart, *i.e.*, mounting the same file system under the same mount point for the second time, will invariably fail and trigger a different set of actions (along the failure path inside the syscall). Consequently, we exclude those syscalls (Appendix A) for this experiment.

Action	Re-ex. success	Action	Re-ex. success
Alloc ($\mathcal{A}_a$)	68.07%	Dealloc ($\mathcal{A}_d$)	42.92%
Val Read ($\mathcal{A}_{vr}$)	38.91%	Val Write ($\mathcal{A}_{vw}$)	32.91%
Ptr Read ($\mathcal{A}_{pr}$)	38.18%	Ptr Write ($\mathcal{A}_{pw}$)	56.27%
Idx Read ($\mathcal{A}_{ir}$)	29.37%	Idx Write ($\mathcal{A}_{iw}$)	18.49%
	Overall	54.68%	

Table 3: Re-execution success of all event types on Linux 5.17

We measure ACTOR's re-execution success averaged over 5 runs of 24 hours each. Table 3 shows the result of this experiment. We see that re-execution works best for $\mathcal{A}_a$, $\mathcal{A}_d$, and $\mathcal{A}_{pw}$, while $\mathcal{A}_{iw}$ and $\mathcal{A}_{ir}$ exhibit a lower success rate.

We can intuitively see why $\mathcal{A}_a$ has a re-execution success rate higher than almost all the other action types. Imagine, there is a list in the kernel where an allocation gets stored. When the $\mathcal{A}_a$ dart was recorded, it got stored at slot 1. The recorded $\mathcal{A}_r$ dart reads from slot 1, too. When both darts are re-executed, $\mathcal{A}_a$ still succeeds, but now the allocation gets stored in slot 0, while the $\mathcal{A}_r$ dart still tries to read from slot 1—which makes it fail.

The success rate of $\mathcal{A}_a$ is positively correlated with $\mathcal{A}_{pw}$, which also has a high success rate. When an allocation happens, the allocated region gets stored in a pointer variable, thus generating an $\mathcal{A}_{pw}$ action. Since the success rate of $\mathcal{A}_a$ is high, and a fraction of those allocations will be stored in heap pointers, they will immediately generate $\mathcal{A}_{pw}$ actions—leading to a high success rate.

$\mathcal{A}_{iw}$ has the lowest success rate. Recall that we only record the first write, per allocation, per syscall. Such a write would mostly be an "initializing" write, which *should* happen only once, unless, of course, there is a bug. We use $\mathcal{A}_{iw}$ darts in the OOB-Index template, where we attempt to repeat ($\mathcal{A}^*_{iw}$) this action. In a bug-free execution, all but the first $\mathcal{A}_{iw}$ attempt would fail, thus lowering its success rate.

RQ2: Darts of almost all action types have acceptable re-execution success rates for ACTOR's strategy to work.

5.3 Shared accesses (RQ3)

The core idea of ACTOR is based on the insight that a set of actions – performed on the same memory buffer in a certain order – is required to expose a bug. To generate input programs that invoke such actions on the same objects (allocations), our system performs *group merging* (Section 3.1). Thus, if ACTOR's *merging* strategy is effective, it should generate groups with related darts that operate on a common memory buffer. Then, during *program synthesis*, since ACTOR chooses darts from these groups, the darts should result in actions that generate shared memory accesses.

The amount of shared accesses is our proxy metric to understand the efficacy of the group merging algorithm. The more shared accesses we can trigger, the better we consider our merging strategy to be.

We run both ACTOR and SYZKALLER+ for 24 hours to measure the shared accesses generated by each fuzzer, for each kernel subsystem. We excluded the same 14 syscalls that we identified in **RQ2**. Shared access is measured after the *dart grouping* phase. Recall that a group contains darts that operate on a common allocation (buffer), by definition. Therefore, for a group with d darts, we count d shared accesses. For groups with only one allocation dart, we exclude that group from counting. To gain subsystem-specific insight, we map each dart to a kernel subsystem. We first assign each dart to a subsystem, which is determined by the location of the instruction that triggered the respective action. A group's subsystem is then taken to be the one to which the majority of its darts belong.

Subsystem	ACTOR	SYZKALLER+	Improvement
arch/	389,978	321,947	21.13%
block/	158,012	168,602	−6.28%
certs/	0	0	0.00%
crypto/	29,405	41,628	−29.36%
drivers/	1,068,698	775,113	37.88%
fs/	4,222,386	4,733,901	−10.81%
ipc/	132,340	141,906	−6.74%
kernel/	3,684,928	2,734,836	34.74%
lib/	1,408,382	659,223	113.64%
mm/	132,374	117,369	12.78%
net/	6,178,633	3,409,346	81.23%
security/	3,020,899	2,925,410	3.26%
sound/	261,289	224,960	16.15%
total	20,687,324	16,254,239	27.27%

Table 4: Shared accesses of ACTOR and SYZKALLER+ per subsystem

The number of subsystem-specific shared memory accesses generated by both fuzzers are presented in Table 4. ACTOR triggers 27.27% more shared accesses than SYZKALLER+ across all subsystems. Interestingly, ACTOR significantly underperforms with respect to SYZKALLER+ in the `crypto`, `ipc`, and `block` subsystems. This is because ACTOR can only start using groups for program synthesis once we have darts for all the action types that the chosen strategy uses. Those subsystems being small, it takes quite some time to accumulate enough darts of all required types. That is why the *program synthesis* starts delayed and also happens at a lower rate, which together hurts ACTOR's performance.

RQ3: ACTOR achieves 27.27% more shared accesses than SYZKALLER+ across all subsystems.

5.4 Bug-inducing program generation (RQ4)

A coverage-guided fuzzer such as SYZKALLER lacks specific strategies to create inputs that target specific bug patterns. The goal of this experiment is to measure if SYZKALLER generates likely bug-inducing programs.

Since we rely on ACTOR's groups to compute this metric, we compare ACTOR with SYZKALLER+, which generates ACTOR-style groups but uses SYZKALLER's synthesis algorithm. We record 10% of all programs generated by both ACTOR and SYZKALLER+ during 24-hour runs, along with the discovered groups. We then count the number of programs that conform to one of our bug templates.

For example, to see if a program $\mathcal{P} = \{s_1, s_2, s_3, s_4, ...\}$ matches a bug template $bt = A_1 \rightarrow A_2^* \rightarrow A_3$ (where A_is are action types), **(i)** we consider all possible continuous subsequences of syscalls of $\mathcal{P}$ to which bt can possibly be expanded, and **(ii)** for each subsequence, we check if all syscalls s_i appear in any one of the recorded groups with the action types expected by bt. Considering $\mathcal{P}$'s first subsequence $\{s_1, s_2, s_3\}$ which bt could be expanded to, we would count the number of groups containing $\{s_1, s_2, s_3\}$ syscalls (darts) having $\{A_1, A_2, A_3\}$

action types, respectively. However, while finding darts in groups, we disregard the syscall arguments, and perform the matching only on the basis of syscall names. The rationale is that, even if a syscall appears with a different set of arguments in a group, it is hard to determine if it still triggers the intended action. We relaxed the matching criteria since any resulting imprecision affects both fuzzer versions in the same way.

Table 5 shows the number of programs that match a bug template, for both fuzzers. The *improvement* column shows, for each template and finally across all templates, the factor by which ACTOR generates more bug-inducing programs than SYZKALLER+. It can be seen that ACTOR outperforms SYZKALLER+ on all patterns. In addition, ACTOR does not improve much for IF, ML-1 and UR. Those templates are fairly short and simple, hence, it is relatively easy for a program to match with those templates. For instance, IF requires one single $\mathcal{A}_d$ syscall, which is not challenging for SYZKALLER+ to generate. For longer and more complex patterns, such as

Strategy	ACTOR	SYZKALLER+	Improvement
Use After Free (UAF)	2,085,492	92,844	22.46
Double Free (DF)	2,266,692	79,440	28.53
Out of Bounds (OOB-1)	517,092	24,702	20.93
Out of Bounds (OOB-2)	422,298	11,160	37.84
Uninitialized Read (UR)	5,906,130	1,960,104	3.01
Null Ptr Deref (NPD)	7,468,788	652,764	11.44
Invalid Free (IF)	26,296,746	22,583,958	1.16
Memory Leak (ML-1)	215,840,400	180,381,540	1.20
Memory Leak (ML-2)	986,856	45,486	21.70
Total	261,790,494	205,831,998	1.27

Table 5: Bug-inducing programs generated by ACTOR *vs.* SYZKALLER+

OOB-1/2, DF, and ML-2, ACTOR significantly outperforms SYZKALLER+.

RQ4: ACTOR generates significantly more programs that target specific (and interesting) bug patterns compared to SYZKALLER+.

5.5 Syscall affinity (RQ5)

Unlike traditional coverage-guided fuzzers such as SYZKALLER, ACTOR performs *group merging* (during *action mining*) to discover relationships among darts (and syscalls). In this section, we explore if the *merging* step discovers relations that SYZKALLER does not find.

Recall that SYZKALLER maintains a *choicetable ct* to choose the next syscall during program synthesis. The *choicetable* is a two-dimensional matrix that holds a *weight* for each pair of syscalls s_i and s_j. SYZKALLER uses this *weight* value to determine the likelihood of placing those two syscalls together in an input program (this likelihood is called *affinity*). The *choicetable* is computed based on both static and dynamic feedback. The static feedback relies on the argument and return types of syscalls. If s_i and s_j share arguments of the same type, the corresponding static weight will be higher. The dynamic feedback is based on the number of times that two syscalls appear together in a program that is part of the fuzzing corpus. Programs are added to the corpus if they find new coverage. Hence, the dynamic component increases the weight of a syscall pair when the corresponding syscalls frequently appear together in the corpus. In summary, when the weight for a syscall pair in the *choicetable* is above average, we can assume that SYZKALLER has identified some type of relationship between the two corresponding syscalls.

When ACTOR decides to merge two groups g_1 and g_2, it basically infers a relationship between every pair of syscalls $(s_1,s_2) \in g_1 \times g_2$. For every pair (s_1,s_2), we can then check how likely it would be for SYZKALLER to put these two syscalls together in a program. We do this by consulting its *choicetable*. Specifically, we compute the average *weights* $av_1 = \texttt{Avg}(ct[s_1])$ and $av_2 = \texttt{Avg}(ct[s_2])$ across all pairs of syscalls that contain s_1 or s_2. We call a merge *unlikely* if the probability of s_1 appearing in a program next to s_2 (or vice versa) is less than average, *i.e.*, either $ct[s_1][s_2] < avg_1$ or $ct[s_2][s_1] < avg_2$.

We ran ACTOR for 24 hours on Linux kernel 5.17, and sampled 10% of all merges. Out of 36,649 merges, 8,082 (22.05%) were considered *unlikely*. This shows that ACTOR is able to infer relations through *merging* that SYZKALLER would not consider. Note that the improvement in the learned relations does not have an impact on the produced coverage, because ACTOR replays recorded darts. The replay, even if successful, exercises no new path, thus not contributing to overall coverage.

RQ5: Dart merging enables ACTOR to learn syscall relations that SYZKALLER does not discover.

Case study: We describe a relationship between two syscalls that SYZKALLER missed, but ACTOR discovered. Specifically, we look at the example of the two syscalls `pipe2` and `close`.

When `pipe2` is invoked with certain arguments, it will call the function `alloc_pipe_info`. This function has a local variable `pipe` of type `struct pipe_inode_info*`, which points to a memory object. At some point in the function, the kernel allocates another memory object and writes that (new) address into `pipe->bufs`. This write is a heap pointer write action (according to our definitions in Section 3.1), and it is recorded by ACTOR. The syscall `close`, when invoked on a file descriptor that is part of a pipe, will reach the function `free_pipe_info`. In this function, the kernel will free the `struct pipe_inode_info*` associated with the pipe. This deallocation will be recorded by ACTOR as a heap deallocation action. It is clear that these two syscalls can access the same memory object (the `struct pipe_inode_info*`) and, therefore, they have a relation.

We performed an experiment to see whether SYZKALLER discovers this relation by checking for a high priority score for the two syscalls in the *choicetable*. We also determine if ACTOR discovers the relationship. For this experiment, we used the Linux kernel 5.17. ACTOR found the relationship between `pipe2` and `close` in less than one hour. On the other hand, at the time when ACTOR discovered the relationship between these two syscalls, SYZKALLER was, according to our definition, unlikely to place these two syscalls together in a program.

5.6 Comparison to the state-of-the-art (RQ6)

We compare ACTOR with respect to the state-of-the-art fuzzers in terms of coverage and bugs found over a period of 24 hours.

SYZKALLER [18], HEALER [58] and MOONSHINE [45] are the closest to our work, which have the following things in common: **(a)** they learn relations between syscalls, and **(b)** they are general-purpose fuzzers, *i.e.*, they do not target specific bug-classes/subsystems. Likewise, **(a)** ACTOR's dart grouping and merging strategies implicitly learn relations between darts, and **(b)** ACTOR does not target any specific class of bugs like race [30], or any a subsytem like file-system [34, 64]. Even though the current prototype only offers bug templates targeted towards memory errors, it is one of the most dominant bug class the covers many different sub-types. In addition, ACTOR can very well be extended (Section 6) to other classes of bugs, too.

MOONSHINE distills a large corpus of seeds down to a much smaller, yet effective one. Information theoretically speaking, the resulting corpus has a higher entropy. Since all other fuzzers started from an empty set of seeds, it led to much of the fuzzing cycles being spent to build up the "knowledge" that MOONSHINE had the leverage to start with right from the beginning. In fact, the seeds that MOONSHINE used came from multiple sources, and potentially a longer fuzzing campaign. Since MOONSHINE works only with old kernel versions [15] not supported by ACTOR, we resorted to cross-pollination [3] as the best-effort approach to compare against their tool. Cross-pollination is a standard practice in the fuzzing community, and ideal for cases like this where two different libraries (in our cases, two different versions of the kernel) accept a common data format. Though kernel APIs may change across versions, we do not expect the change to be significant enough (with respect to the total number of syscalls) so that it entirely invalidates a corpus from one version of the kernel to be used with another. Therefore, we used the seeds that the authors of MOONSHINE used in their experiments with the kernel that ACTOR supports. For HEALER, we used its public version for our evaluation (but we do note that there is also a private version that the authors of HEALER used for their experiments).

The coverage and the number bugs found by different fuzzers are shown in Figure 4 and Figure 5, respectively. SYZKALLER achieves highest coverage, followed closely by ACTOR (0.42% less) and then MOONSHINE (6.39% less). HEALER performs significantly worse than all other tools. During the same period, SYZKALLER found 9 unique crashes, ACTOR and MOONSHINE found 10 each, and HEALER found none. ACTOR had 8 crashes common with SYZKALLER and 6 with MOONSHINE. ACTOR found 2 bugs not found by any

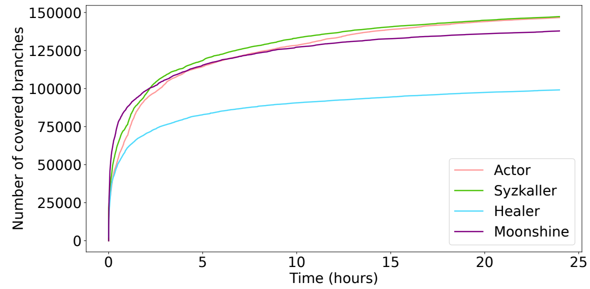

Figure 4: Coverage attained by ACTOR, SYZKALLER, HEALER, and MOONSHINE over 24 hours

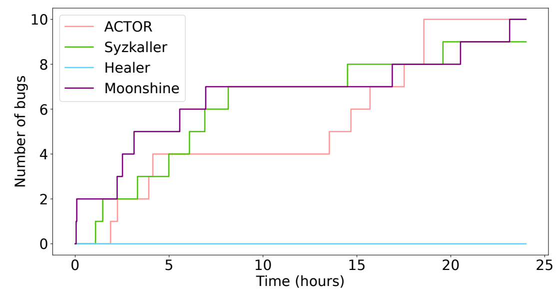

Figure 5: Bugs found by ACTOR, SYZKALLER, HEALER and MOONSHINE over 24 hours

other tool. This reinforces our hypothesis that merely covering more code, which most fuzzers optimize for, is not enough for triggering all bugs. Interestingly, ACTOR found 5 memory corruption bugs, while SYZKALLER as well as MOONSHINE found the same 3. This can be explained by the fact that ACTOR's templates specifically target a range of memory bugs, which gives it an edge over other fuzzers. Additionally, MOONSHINE is the only fuzzer that found crashes of type "INFO", which points to potential issues in the kernel logic, whereas ACTOR found more memory errors, which is what ACTOR 's templates are tailored to.

RQ6: ACTOR finds bugs that other fuzzers cannot, while achieving comparable coverage.

6 Discussion and Limitation

Support for more action types. Currently, ACTOR only supports action types that are related to the kernel heap. This means that ACTOR is unable to target certain bugs, for example, those involving global variables and reference counters. Additional action types can be supported by extending the *actrack* module. These additions would then allow us to write bug templates that target other classes of bugs, such as refcount mismatches.

In addition to the simple action types that we currently implement, ACTOR could also benefit from more complex ones. One such example would be an action that is based on the *points-to* relation between two memory objects. For example, there could be a structure S_1 with a pointer field $S_1.p$, which points to another structure S_2. Identifying such connections would allow for additional, or even more complex strategies. For instance, we could improve the ML-2 template by substituting the $\mathcal{A}_{pw}$ action type with this specialized type. One way of supporting complex actions would be to develop a more sophisticated *semantic labeling* phase, and refine generic actions to more specialized types, in the same way ACTOR does in its current implementation for pointer and array index accesses. Alternatively, instead of using the *actrack* module (that indiscriminately intercepts a broad range of actions without knowing their semantics), we can selectively instrument the kernel by writing an LLVM transformation pass.

Inseparable/connected actions. When synthesizing fuzzer programs, ACTOR assumes that it is possible to arbitrarily reorder any action. This is not always the case. Consider a syscall that triggers two actions: $\mathcal{A}_a$ and $\mathcal{A}_{vw}$. We would currently add these actions to a group as separate and independent darts d_a and d_{vw}. Normally, actions can be reordered fairly easily when they come from different syscalls. If they come from the same syscall, however, they may or may not be reordered (depending on the kernel state). In our example, using the d_a dart for the uninitialized read (UR) template may not be successful. As both the allocation and the access actions are performed by the same syscall, they are likely to get invoked one after the other, causing the memory to get initialized before the read happens. There is no easy way to determine if actions such as $\mathcal{A}_a$ and $\mathcal{A}_{vw}$ are always ordered. A static analysis or under-constrained symbolic execution-based technique [33] could help. However, in that case, such an analysis needs to be integrated within the fuzzing loop, which would drastically slow down the input generation process.

False negatives of semantic labeling. Our *semantic labeling* is *unsound*. As explained in Section 3.1, based on our observation, we make the assumption that the kernel heap allocations are structures, because it would be unusual to allocate a primitive data type on the heap. However, there could still be cases where this assumption does not hold. In those cases, we will fail to detect an $\mathcal{A}_{p*}/\mathcal{A}_{i*}$ action, thus, mislabeling the action type.

7 Related Work

In this section, we discuss prior research efforts on detecting vulnerabilities in the OS kernels.

Structure/relation-learning-aided fuzzers. Such fuzzers attempt to infer the shape of syscall arguments, as well as inter-syscall relations, to **(i)** invoke syscalls with well-formed arguments, and **(ii)** to order syscalls meaningfully to expose deeper functionality. `Syzlang` is a domain-specific language that can encode complex argument types for syscalls. To learn dependence relations between pairs of syscalls, both SYZKALLER [18] and HEALER [58] rely on manually defined syscall descriptions written in `Syzlang`, as well as on coverage-guided feedback for dynamic reasoning. MOONSHINE [45] leverages static program analysis to infer implicit dependencies, and a trace-based analysis for explicit dependencies, which is similar to IMF's [28] approach. HFL [33] performs symbolic execution to infer complex syscall sequences and to construct nested syscall arguments. To recover valid commands and the argument structure of the `ioctl` interfaces, DIFUZE [24] employs a combination of static, inter-procedural, path-sensitive analysis and range analysis. NTFUZZ [23] designed a bottom-up, summary-based algorithm to infer the types of syscall arguments which is captured by their abstract domain. While all these approaches strive to improve (code) coverage by synthesizing better programs, unlike ACTOR, none of them make any particular effort to craft inputs that are specifically tailored to trigger bugs.

Subsystem-targeted fuzzing. While syscall fuzzers mimic adversarial attacks from user-land, driver fuzzers assume a stronger attack model by considering peripheral devices to be malicious. To enable fuzzing from the peripheral side, they either use a physical device [56], a host-forwarded physical device [39, 59], a symbolic device [36, 51], a virtual device [57, 66], or in-process IO interception from a library OS [29]. Moreover, fuzzers have been developed for specific classes of kernel drivers, such as USB [32,47] and WiFi [21,42], or for specific subsystems, such as the file system [34,43,63,64] and parsers [38]. Unlike these fuzzers, ACTOR is neither a peripheral surface fuzzer, nor targets any specific subsystem.

Enhancing bug-finding capabilities. While most kernel fuzzers work with open-source OSes, few [35, 46] work with Commercial-Off-The-Shelf (COTS) OSes, too. For example, KAFL [50] supports COTS OSes and improves fuzzing throughput by using Intel Processor Trace, for collecting near-zero overhead, OS-independent, coverage feedback. Some fuzzers adopt optimized strategies to trigger hard-to-trigger bugs, for example, race conditions [25–27, 30]. ACTOR relies on source code for semantic labeling, therefore can not work with COTS OSes. Moreover, rather than focusing on one specific bug type, ACTOR supports diverse classes of bugs due to its template-guided approach. UAFL [60] uses *operation sequence* as the coverage feedback to guide the program mutation. STATEFUZZ [65] models the kernel state as a set of state-variables. The fuzzer uses a combination of code coverage and state changes as guidance for the exploration of the kernel state space. Since ACTOR can precisely control the actions by controlling syscall invocations, therefore, unlike these fuzzers which pass on new feedback signals, ACTOR leverages actions for program generation, guided by bug templates.

Program synthesis. Given a specification, program synthesis is the technique to automatically generating valid programs. To this date, synthesis has been successfully used in many different contexts. A common application of program synthesis is in program repair. ANGELIX [41], SEMFIX [44], and

DIRECTFIX [40] use semantic information obtained through symbolic execution and constraint solving to synthesize the correct version of a buggy program. Tools like GENPROG [37], RSREPAIR [48] synthesize bug-free programs by traversing the search space of possible fixes, and validating them against test cases. ACS [62] synthesizes program conditions by first selecting candidate variables through a ranking technique, and then applying necessary predicates to them found in other similar contexts. SEQUENCER [22] combines a machine learning technique called sequence-to-sequence learning with the construction of an abstract buggy context to generate one-line patches. Sketching systems [52–55] consume a high-level description of an algorithm, which is then instantiated using program synthesis. Systems such as, PSKETCH [53], SKETCH [55], STREAMBIT [54] use counter-example-guided inductive synthesis. Code completion, another application of program synthesis, is dominated by machine-learning-based solutions [7, 16, 49, 61]. Lastly, SOUFFLÉ [31] leverages program synthesis to generate static analyzers capable of statically analyzing software products. Unlike previous research, ACTOR applies template-guide synthesis to the domain of kernel fuzzing in order to generate fuzzer programs.

8 Conclusion

In this paper, we presented ACTOR, a novel program (input) generation strategy for kernel fuzzing. Our *action-guided* synthesis technique is complementary to the traditional coverage-guided strategy that attempts to maximize code coverage.

ACTOR generates potentially bug-triggering programs by following templates written in a domain-specific language (DSL). Action-guided program generation is effective, as ACTOR discovered 41 previously unknown bugs in two well-tested and actively-patched long-term releases of the Linux kernel as well as its latest stable release version.

Acknowledgement

We thank the anonymous reviewers for their valuable suggestions and comments. We would also like to thank Fabio Pagani for his valuable comments and input to improve our paper. This material is based on research sponsored by DARPA under agreement number N6600120C4031. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

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

[2] An overview of live kernel patching. https://ubuntu.com/blog/an-overview-of-live-kernel-patching.

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

[4] CVE-2019-11815. https://www.trendmicro.com/en_us/research/19/e/cve-2019-11815-a-cautionary-tale-about-cvss-scores.html.

[5] CVE-2022-0185. https://blog.aquasec.com/cve-2022-0185-linux-kernel-container-escape-in-kubernetes.

[6] CVE-2022-32250. https://www.openwall.com/lists/oss-security/2022/06/03/1.

[7] Github Copilot. https://github.com/features/copilot/.

[8] https://github.com/google/kmsan. https://www.kernel.org/doc/html/latest/dev-tools/kasan.html.

[9] Inter-VM shared memory. https://github.com/qemu/qemu/blob/master/docs/specs/ivshmem-spec.txt.

[10] KASAN: Kernel Address Sanitizer. https://www.kernel.org/doc/html/latest/dev-tools/kasan.html.

[11] KCSAN: Kernel Concurrency Sanitizer. https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html.

[12] Linux kernel BUG_ON assertion. https://kernelnewbies.org/FAQ/BUG.

[13] Linux kernel fault injection. https://docs.kernel.org/fault-injection/index.html.

[14] Linux kernel runtime locking correctness validator. https://www.kernel.org/doc/Documentation/locking/lockdep-design.txt.

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

[16] OpenAI Codex. https://openai.com/blog/openai-codex.

[17] Syzbot: Continuous Linux kernel fuzzing. https://syzkaller.appspot.com.

[18] Syzkaller: An unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller.

[19] The LLVM Compiler Infrastructure. https://llvm.org.

[20] UBSAN: Undefined Behavior Sanitizer. https://www.kernel.org/doc/html/latest/dev-tools/ubsan.html.

[21] Wi-fi advanced fuzzing. https://www.blackhat.com/presentations/bh-europe-07/Butti/Presentation/bh-eu-07-Butti.pdf.

[22] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 2021.

[23] J. Choi, K. Kim, D. Lee, and S. K. Cha. Ntfuzz: Enabling type-aware kernel fuzzing on windows with static binary analysis. In *Proc. IEEE Symposium on Security and Privacy*, 2021.

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

[25] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *Proc. USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[26] P. Fonseca, R. Rodrigues, and B. B. Brandenburg. Ski: Exposing kernel concurrency bugs through systematic schedule exploration. In *Proc. USENIX Symposium on Operating Systems Design and Implementation*, 2014.

[27] S. Gong, D. Altinbüken, P. Fonseca, and P. Maniatis. Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis. In *Proc. ACM SIGOPS Symposium on Operating Systems Principles*, 2021.

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

[29] F. Hetzelt, M. Radev, R. Buhren, M. Morbitzer, and J.-P. Seifert. Via: Analyzing device interfaces of protected virtual machines. In *Proc. Annual Computer Security Applications Conference*, 2021.

[30] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin. Razzer: Finding kernel race bugs through fuzzing. In *Proc. IEEE Symposium on Security and Privacy*, 2019.

[31] H. Jordan, B. Scholz, and P. Subotic. Soufflé: On synthesis of program analyzers. In *Proc. International Conference on Computer Aided Verification*, 2016.

[32] D. Kierznowski. Badusb 2.0: Exploring usb man-in-the-middle attacks.

[33] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee. Hfl: Hybrid fuzzing on the linux kernel. In *Proc. The Network and Distributed System Security Symposium*, 2020.

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

[35] 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 *Proc. USENIX Annual Technical Conference*, 2017.

[36] V. Kuznetsov, V. Chipounov, and G. Candea. Testing Closed-Source binary device drivers with DDT. In *Proc. USENIX Annual Technical Conference*, 2010.

[37] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 2012.

[38] D. Maier, B. Radtke, and B. Harren. Unicorefuzz: On the viability of emulation for kernelspace fuzzing. In *Proc. USENIX Workshop on Offensive Technologies*, 2019.

[39] D. Maier and F. Toepfer. Bsod: Binary-only scalable fuzzing of device drivers. In *Proc. International Symposium on Research in Attacks, Intrusions and Defenses*, 2021.

[40] S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *Proc. International Conference on Software Engineering*, 2015.

[41] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proc. International Conference on Software Engineering*, 2016.

[42] M. Mendonça and N. Neves. Fuzzing wi-fi drivers to locate security vulnerabilities. In *Proc. European Dependable Computing Conference*, 2008.

[43] J. Mohan, A. Martinez, S. Ponnapalli, P. Raju, and V. Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *Proc. USENIX Symposium on Operating Systems Design and Implementation*, 2018.

[44] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proc. International Conference on Software Engineering*, 2013.

[45] S. Pailoor, A. Aday, and S. Jana. MoonShine: Optimizing OS fuzzer seed selection with trace distillation. In *Proc. USENIX Security Symposium*, 2018.

[46] J. Pan, G. Yan, and X. Fan. Digtool: A virtualization-based framework for detecting kernel vulnerabilities. In *Proc. USENIX Security Symposium*, 2017.

[47] H. Peng and M. Payer. USBFuzz: A framework for fuzzing {USB} drivers by device emulation. In *Proc. USENIX Security Symposium*, 2020.

[48] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *Proc. International Conference on Software Engineering*, 2014.

[49] V. Raychev, M. T. Vechev, and E. Yahav. Code completion with statistical language models. 2014.

[50] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz. kafl: Hardware-assisted feedback fuzzing for os kernels. In *Proc. USENIX Security Symposium*, 2017.

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

[52] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Sketching stencils. In *Proc. ACM SIGPLAN Symposium on Programming Language Design and Implementation*, 2007.

[53] A. Solar-Lezama, C. G. Jones, and R. Bodík. Sketching concurrent data structures. In *Proc. ACM SIGPLAN Symposium on Programming Language Design and Implementation*, 2008.

[54] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu. Programming by sketching for bit-streaming programs. In *Proc. ACM SIGPLAN Symposium on Programming Language Design and Implementation*, 2005.

[55] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[56] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *Proc. The Network and Distributed System Security Symposium*, 2019.

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

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

[59] S. M. S. Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. A. Sani, and Z. Qian. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In *Proc. USENIX Security Symposium*, 2018.

[60] H. Wang, X. Xie, Y. Li, C. Wen, Y. Li, Y. Liu, S. Qin, H. Chen, and Y. Sui. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. 2020.

[61] Y. Wang, W. Wang, S. Joty, and S. C. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proc. Conference on Empirical Methods in Natural Language Processing*, 2021.

[62] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang. Precise condition synthesis for program repair, 2017.

[63] M. Xu, S. Kashyap, H. Zhao, and T. Kim. Krace: Data race fuzzing for kernel file systems. In *Proc. IEEE Symposium on Security and Privacy*, 2020.

[64] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim. Fuzzing file systems via two-dimensional input space exploration. In *Proc. IEEE Symposium on Security and Privacy*, 2019.

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

[66] W. Zhao, K. Lu, Q. Wu, and Y. Qi. Semantic-informed driver fuzzing without both the hardware devices and the emulators. In *Proc. The Network and Distributed System Security Symposium*, 2022.

A Discarded syscalls

Syscall name	Syscall name
lsetxattr$security_selinux	openat
mount	chown
ioctl$BLKTRACESETUP	add_key$keyring
openat$procfs	ioctl$sock_SIOCGIFINDEX_80211
syz_mount_image$tmpfs	syz_clone3
syz_mount_image$iso9660	syz_mount_image$vfat
syz_mount_image$ext4	syz_open_dev$sg

Table 6: 14 syscalls that are bad for re-execution.