SNOWPLOW: Effective Kernel Fuzzing with a Learned White-box Test Mutator

Sishuai Gong
Purdue University
West Lafayette, IN, USA

Rui Wang
Purdue University
West Lafayette, IN, USA

Deniz Altınbüken
Google DeepMind
Mountain View, CA, USA

Pedro Fonseca
Purdue University
West Lafayette, IN, USA

Petros Maniatis
Google DeepMind
Mountain View, CA, USA

Abstract

Kernel fuzzers rely heavily on program mutation to automatically generate new test programs based on existing ones. In particular, program mutation can alter the test's control and data flow inside the kernel by inserting new system calls, changing the values of call arguments, or performing other program mutations. However, due to the complexity of the kernel code and its user-space interface, finding the effective mutation that can lead to the desired outcome such as increasing the coverage and reaching a target code location is extremely difficult, even with the widespread use of manually-crafted heuristics.

This work proposes SNOWPLOW, a kernel fuzzer that uses a learned white-box test mutator to enhance test mutation. The core of SNOWPLOW is an efficient machine learning model that can learn to predict promising mutations given the test program to mutate, its kernel code coverage, and the desired coverage. SNOWPLOW is demonstrated on argument mutations of the kernel tests, and evaluated on recent Linux kernel releases. When fuzzing the kernels for 24 hours, SNOWPLOW shows a significant speedup of discovering new coverage (4.8×~5.2×) and achieves higher overall coverage (7.0%~8.6%). In a 7-day fuzzing campaign, SNOWPLOW discovers 86 previously-unknown crashes. Furthermore, the learned mutator is shown to accelerate directed kernel fuzzing by reaching 19 target code locations 8.5× faster and two additional locations that are missed by the state-of-the-art directed kernel fuzzer.

CCS Concepts: • **Security and privacy → Operating systems security**; • **Software and its engineering → Software defect analysis**.

Keywords: Kernel fuzzing, Operating systems reliability and security, Software testing and debugging

This work is licensed under a Creative Commons Attribution 4.0 International License.

ASPLOS '25, Rotterdam, Netherlands
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1079-7/25/03
https://doi.org/10.1145/3676641.3716019

ACM Reference Format:
Sishuai Gong, Rui Wang, Deniz Altınbüken, Pedro Fonseca, and Petros Maniatis. 2025. SNOWPLOW: Effective Kernel Fuzzing with a Learned White-box Test Mutator . In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3676641.3716019

1 Introduction

Kernel fuzzers [26, 28, 30] test the kernel by running user-space programs that invoke kernel APIs (i.e., system calls). To automatically generate new kernel tests, modern fuzzers rely heavily on mutation techniques [8, 34, 46] that create new tests by progressively applying minor changes—such as new system call invocations and new argument values—to a base test program. By focusing on minor and controlled changes at each test generation, fuzzers can systematically discover new ways to invoke system calls in the base test and increase the likelihood of reaching new kernel branches [32, 53] missed by the base test but reachable with slightly different initial conditions (e.g., different system-call arguments), therefore testing the kernel more comprehensively.

However, effectively mutating the base test to exercise new kernel code remains challenging due to a dauntingly large search space rooted in the complexity of the kernel APIs (i.e., system calls). For mutations that try to insert new system calls, hundreds of system calls are available to choose from, but only a few of them are effective in changing the behavior of existing calls [9, 35, 46], and such control dependencies are often implicit and under-documented [1, 27, 38, 46]. For argument mutations, although system calls may seem to input a limited number of arguments at first glance, each argument can further encode several sub-level arguments [32, 33] via data structures such as nested `struct`, different types, different memory stances, etc. Together, they contribute to a non-trivial search space—a kernel test on average contains more than 60 arguments available for mutation (§5.1).

Modern kernel fuzzers employ heuristics and randomness [15, 21, 27, 38, 41, 46, 52, 53] to explore the vast argument search space. For example, Syzkaller [26]—the state-of-the-art kernel fuzzer–uses heuristics to assign weights

to different mutation choices (e.g., which system calls to insert) and then makes decisions semi-randomly, favoring options prioritized by heuristics while occasionally choosing de-prioritized ones in case heuristics are imperfect. This strategy allows Syzkaller to rapidly generate new tests, but it bears efficiency challenges in finding effective mutations in the extensive search space. According to our measurements (§5.1), Syzkaller on average can only discover 44 new tests that trigger kernel branches missed by the base test after 1000 argument mutations of the base.

White-box mutation techniques [3, 17, 20] are promising to fundamentally address the challenges of test mutation in complex systems, like kernels. For every test to mutate, the white-box mutator performs precise analysis of the executed kernel code path, solves the constraints of the uncovered branches and guides the test mutations accordingly. Building such a mutator typically requires heavyweight methods such as symbolic execution [5, 10, 12, 39, 55]. However, the high cost and poor scalability of such approaches prevent kernel fuzzers from using them as a first-option mutator, opting instead to use them as a fallback mechanism to existing heuristics [11, 29, 44, 55]. For example, HFL [32], which employs symbolic execution to assist kernel test generation, only invokes the symbolic engine when certain kernel branches are repeatedly missed by Syzkaller, at which point significant testing resources were already wasted.

This paper proposes Snowplow, a kernel fuzzer that uses a learned white-box mutator to mutate kernel tests. Snowplow pursues the direction of *symbolizing* the kernel test using machine learning. The core of Snowplow is a new ML model architecture called Program Mutation Model (*PMM*). *PMM* takes the kernel test (i.e., a user-space program), its kernel coverage (kernel code blocks triggered by that test), and the desired target coverage (i.e., a block not covered yet) as input, and predicts mutations that lead to the target coverage. *PMM* bridges the gap between the precise successful mutations that kernel fuzzers discover through thousands of trials and the lack of effective methods to memorize and generalize these mutation strategies, making an accurate and scalable white-box mutator for the kernel test feasible.

We present *PMM* focusing on a case study of argument mutations specifically, one of the mutation types that increase coverage the most in existing gray-box fuzzers. We train *PMM* using a dataset of successful argument mutations discovered on the real-world kernel, on the task of identifying which arguments, when mutated, would lead the test to reach the desired target coverage. Then, we build Snowplow to orchestrate the argument mutations of a pre-existing kernel fuzzer, while leaving other mutation types to their existing handcrafted mutations and heuristics. Although we demonstrate the value of *PMM* in the context of argument mutations, it could potentially be used to learn most test mutation types used in modern kernel fuzzers (§6), and is composable with diverse types of hand-crafted heuristics.

Snowplow is evaluated on recent *stable* Linux kernel releases. First, when fuzzing the same kernel we trained on, Snowplow can speed up the discovery of edge coverage by 5.2× and achieve 7.0% higher overall edge coverage in a repeated 24-hour side-by-side comparison to Syzkaller. Second, Snowplow can generalize to newer kernels after the one we trained on without re-training or fine-tuning. In a 24-hour comparison, Snowplow on average achieves >4.8× speedup and 7.7%~8.6% higher overall edge coverage. Third, in a 7-day side-by-side fuzzing campaign, Snowplow discovers 86 new crashes (57 with reproducers) in stable kernels, whereas Syzkaller did not; these crashes have not been discovered during the continuous fuzzing efforts of Linux kernels, run by the kernel-developer community. Finally, the learned mutator can significantly improve directed kernel fuzzing, which is a special mode of fuzzing that focuses on testing certain code regions instead of all, e.g., to validate a patch before accepting it. In an experiment that aims to reach certain code locations in the kernel, *PMM* introduces an average speedup of 8.5× on 19 unique target code locations while reaching 2 additional locations that are missed by the state-of-the-art.

This paper makes the following contributions:

- **A design of a machine learning model that learns successful mutations of the kernel test.** Snowplow introduces *PMM*, which inputs both the user-space test program and the kernel code to reason about and learn from the mutation.
- **An implementation of an ML-based white-box kernel test mutator.** This includes discovering and building a dataset of real-kernel test program mutations, implementing and training *PMM*, and integrating it into modern kernel fuzzers.
- **An evaluation of the ML-based white-box mutator.** Snowplow is evaluated to fuzz recent stable Linux kernel releases. Snowplow not only can stably achieve higher coverage on recent kernels, but also helps discover 86 unique new crashes. We conducted a manual analysis on these crashes, analyzed a subset in detail, and reported our analysis results with kernel developers. As of this writing, we diagnosed and reported 7 crashes, of which 4 have been confirmed and 2 have been fixed.

Snowplow is publicly available[1].

2 Motivation

A kernel test is a small *user-space program* that makes system-call invocations. Such tests are generated and executed by kernel fuzzer tools, like Syzkaller, in a fuzzing loop, with a goal of increasing code coverage in the kernel internals or, even better, triggering new crashes indicating a potential unknown bug. Much of the logic in kernel fuzzers is carefully

[1] Snowplow implementation: https://github.com/rssys/snowplow

constructed by system experts with accrued mastery in what mutations of a test might uncover new, suspect behaviors. The general problem we target is using machine learning to *control* such programmatic mutation logic, to achieve high coverage faster and to find more crashing tests.

Figure 1 abstracts the fuzzing logic using Python as pseudocode. The main fuzzing loop (function `fuzz_corpus`) involves picking a test from a corpus of known and previously-executed tests (line 9), mutating the chosen test (line 13), executing the mutated test (line 17), and either reporting a crash if one is triggered (line 19), or measuring the resulting increase in code coverage (line 22). A fuzzing campaign can be undirected—any code block in the control-flow graph of the kernel is desirable to cover (line 4)—or directed to some interesting given targets (line 6), such as code blocks changed recently. Given the result and any new coverage from execution, the fuzzer decides whether to include the newly generated test into the corpus, to mutate the chosen test again, or to pick another test to mutate from the corpus (line 20). Prior work has studied how to optimize these high-level decisions (the control functions `choose_test` and `update_corpus` in our pseudocode) for Syzkaller (e.g., SyzVegas [52]); in this work, we focus on the test-mutation part of this workflow in particular (function `mutate_test`).

When facing an input test to mutate, a specialized mutation engine such as Syzkaller considers three fundamental policy decisions, captured in the pseudocode for `mutate_test` (line 25): (1) what *kind* of mutation to perform (*type selection*, line 28), (2) *where* to apply this mutation on the incoming test (*localization*, line 29), and (3) *how* to perform the chosen mutation type at the chosen location (*instantiation*, line 31). Syzkaller and other mutation engines provide a number of actual mutation strategies (e.g., randomize a flag value, replace an integer with a constant, move a pointer inside a buffer, etc.), and once an instantiation strategy is chosen, it can then be applied to the test under mutation (line 34).

Consider the incoming test at the top of Figure 3. Type selection decided to mutate one of the system call arguments, as opposed to adding or removing a system call, or some other high-level mutation type. Localization decided to perform argument mutation on the second argument of the first system call, as opposed to any other argument of open or `read`. Instantiation decided to turn the `O_CREAT` value in that argument to `O_CREAT|O_RDWR`, as opposed to any other valid flag setting for the open system call.

Mutation experts study and encode the domain of each of these choices. For example, Syzkaller uses a specialized language, `Syzlang` [24], to capture variants of each system call (e.g., for the Linux `mount` system call, there are 12 specialized variants [23], with their own arguments), properties of each system-call argument (e.g., whether it is a buffer or file mode, or a flag in our example), and useful instantiations for each mutation type (e.g., how to mutate an integer, a

```
1  def fuzz_corpus(corpus, choose_test, selector,
2                  localizer, instantiator,
3                  targets):
4    uncovered: set[Block] = cfg_of(KERNEL)
5    covered: set[Block] = {}
6    targets = (uncovered if targets is None
7               else targets)
8    while(not targets <= covered):
9      test, target = choose_test(corpus,
10                                uncovered,
11                                covered,
12                                targets)
13     mutated = mutate_test(test, target,
14                           selector, localizer,
15                           instantiator)
16     try:
17         coverage = execute(mutated)
18     except Crash:
19         report(mutated)
20     update_corpus(corpus, test, mutated,
21                   coverage, uncovered)
22     uncovered = uncovered - coverage
23     covered = covered + coverage
24
25 def mutate_test(
26       test_to_mutate, target,
27       selector, localizer, instantiator):
28   m_type = selector(test_to_mutate, target)
29   location = localizer(test_to_mutate,
30                        target, m_type)
31   instantiation = instantiator(test_to_mutate,
32                                target, m_type,
33                                location)
34   mutated_test = apply_mutation(test_to_mutate,
35                                 m_type,
36                                 location,
37                                 instantiation)
38   return mutated_test
```

Figure 1. Controller logic of a fuzzer. `mutate_test` is illustrated pictorially in Figure 2.

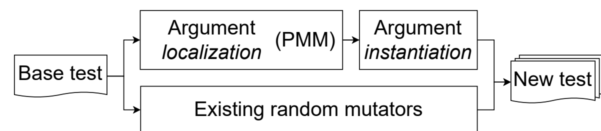


Figure 2. The fuzzing workflow showing a fuzzer and its use for *PMM* as an argument-mutation localizer.

buffer, a file mode, a struct, etc.). Using this "palette" of mutations, experts can also develop heuristics on how to control the search through the palette, to come up with a mutation strategy. These heuristics are captured by the controller functions in the abstraction above, directing type selection,

Incoming Test:

```
1  r0 = open("./file0", O_CREAT, 0777)
2  read(r0, buffer, 42)
```

Mutated Test:

```
1  r0 = open("./file0", O_CREAT|O_RDWR, 0777)
2  read(r0, buffer, 42)
```

Figure 3. Mutation of a simple kernel test.

```
1  ...
2  sendmsg$inet(
3      r1,                      // first arg
4      &(0x7f0000001600)={      // second arg
5          0x0, 0x0,
6          &(0x7f0000001580)=
7              [{&(0x7f0000000280)="db", 0x1}],
8          0x1},
9      0x41)                    // third arg
10 ...
```

Figure 4. Nested arguments in a syz test [47] snippet, showing a second argument (line 4) that is a struct buffer, with 4 fields, the third of which (line 6) is also a nested struct buffer.

localization, and instantiation; a mutator can implement simple controllers (e.g., undirected controllers that ignore the `target` input), or sophisticated white-box controllers such as ours that consider the kernel itself, prior executions, and desirable targets. Syzkaller developers have fixed probabilities by which they choose a mutation type during type selection; in other words, the default `selector` function of syzkaller on Figure 1, line 28 ignores the `target`, flips a biased coin, and returns one of the known types such as `ARGUMENT_MUTATION` or `SYSCALL_INSERTION`. Similarly, Syzkaller considers the arity of each system call when doing localization for argument mutation; the default Syzkaller `localizer` function ignores the `target`, and if `mutation_type` is `ARGUMENT_MUTATION`, it randomly picks an argument from the system call with the largest arity. Such heuristic controllers can require static analysis and the development of complex templates. For instance, SyzDirect [48]—a kernel fuzzer specializing in directed testing [2]—considers what kinds of inputs (i.e., resources) each system call needs, and encodes resource-specific heuristics when mutating upstream system calls; as an illustrative simplification for our example, since there is a `read` call downstream, mutating the mode flag of `open` to create a readable file will be preferred. This kind of logic can be encoded in all three controller functions: `selector`, `localizer`, and `instantiator`.

The purpose of such heuristics is to reduce the size of the search space towards increasing coverage of previously uncovered code more efficiently. This is a well-placed bet: a typical Syzkaller "syz" test contains dozens of arguments, often more than 100; recall that these are not just Linux system-call arguments, but also fields of structs, as well as parameterizations of fixed mechanisms to produce those arguments (e.g., "create a buffer in the heap and fill it with struct `x`, but then shift the pointer 2 bytes into that buffer"). Figure 4 illustrates this, and contains arguments that are numeric constants, structs that initialize fields at explicit memory and stack locations, etc. In our experiments, we see on average more than 60 arguments per test (§5.1). Consequently, when `selector` chooses the `ARGUMENT_MUTATION` type, it has at least a search space of possibly tens of arguments to select from, which is multipled by the instantiation space of the chosen argument's mutation (the composed output domain of `localizer` and `instantiator`); a good choice could lead to orders of magnitude of speedup. This motivated us to focus on argument selection as the target of our work (the implementation of `localizer`, when `selector` returns `ARGUMENT_MUTATION`), since the head-room over semi-random argument selection can be significant. Nevertheless, much of the approach described in this work applies to other mutation types (§6).

Our second motivating consideration comes from the observation that the Linux Kernel, possibly the most-widely distributed software system, is fuzzed continuously, producing many examples of tests, their mutations, and the covered code before and after. This preponderance of data, as well as the promising results from prior work using data-driven approaches to find bugs in software [22, 42, 49], encouraged us to pursue a machine-learning approach to drive the decisions of the test mutator. Rather than using hand-written implementations of `selector`, `localizer` and `instantiator` such as those described above, can we learn from prior observations (of the outputs of those functions, as well as the result of executing the mutated test on Figure 1 line 17) to predict the right choice that ensures a positive reward (i.e., getting a crash on line 19 or a coverage increase on line 22)? In what follows, we show that even a modest deployment of a learned `localizer` function with otherwise unmodified mutators can significantly improve fuzzing efficiency in both the undirected and directed settings.

We revisit our earlier problem definition as follows: in the formulation of Figure 1, design functions `selector`, `localizer`, and `instantiator` so that given the same computational resources (e.g., time, compute, money), the number of reported crashes (Figure 1 line 19) or the achieved coverage (size of `covered` at line 22) surpass those of the baseline (Syzkaller). In the targeted setting (where `targets` is set explicitly in Figure 1 line 6), we seek to reach the targets faster (using fewer resources) than the baseline. In this paper, we focus purely on function `localizer`, in the case where function `selector` has yielded the `ARGUMENT_MUTATION` type, both for directed and undirected fuzzing.

Note that in the computational resources considered we include fixed startup costs (e.g., the cost of data generation

and training of an ML model) as well as per-mutation costs (e.g., in the case of ML, the cost of inference).

3 Design

In this section, we will present a detailed view of a learned argument-mutation localizer, i.e., a ML model trained to predict good arguments to mutate. Our work fits into the existing workflow of a program mutator, such as Syzkaller. We supply the argument localization, i.e., the choice of argument to mutate, so as to cover a particular bit of the kernel code that the test under mutation did not cover, and leave it to the mutator to do the argument instantiation (§2).

Snowplow first collects a dataset of successful argument mutations to train *PMM* (§3.1). We call "successful" a mutation that covered different parts of the kernel from what the base test covered. Then, Snowplow trains a model to predict the location of a successful argument mutation. To capture the structure and semantics of both test programs and kernel code, we use a graph representation of the base test, its coverage, the target we are trying to cover, and the localization result (§3.2). Armed with such training examples, we train *PMM* (§3.3). Finally, we use *PMM* as an argument-mutation localizer in a fuzzer (§3.4).

3.1 Mutation Dataset Generation

The goal of dataset collection is to assemble a large number of examples in which a base test, mutated via an argument mutation, leads to a mutated test with resulting kernel coverage greater than that of the base test; we call that a successful argument mutation. In the absence of an oracle that gives us the perfect argument mutation that increases a test's coverage—after all, it is this oracle we are trying to build—we resort to extensive random mutations using Syzkaller.

Specifically, we start with a *seed corpus* S, a collection of test programs exhibiting broad coverage of the kernel. For each base test $s_i \in S$, we collect the corresponding kernel coverage c_i by executing it, which is a sequence of kernel code blocks executed when the kernel handles s_i. Then, we configure Syzkaller to perform a large number of argument mutations on each test program. So for each mutation j of base test s_i, we ask Syzkaller to localize an argument $a_{i,j}$ to mutate, and then to instantiate the argument mutation into a mutant $s_{i,j}$. We execute each unique $s_{i,j}$ and compare its coverage $c_{i,j}$ to that of the base coverage c_i; if $c_{i,j}$ contains new code block(s) that are not in c_i (i.e., $c_{i,j} \setminus c_i\rangle$), we collect a sample $\langle s_i, c_i, a_{i,j}, c_{i,j} \setminus c_i\rangle$. $a_{i,j}$ may contain one or more arguments, since different mutations sometimes lead to the same new coverage, and we merge those into a single sample. Note that we do not collect the mutated program $s_{i,j}$, since we do not seek to train a model on the instantiation of a mutated argument, and we only collect the newly covered blocks of the mutant.

Recall that our goal is to learn the argument-mutation localization *given a target*. Our samples are generated in the *forward* direction: given a base test and argument selection, find the newly-covered kernel blocks. To "invert" this direction into a predictor, we assemble sensible training examples in which we give a model the base test, its coverage, some targets we desire to cover, and ask for argument selection a_i. We considered a few options: (a) use exactly the new coverage $c_{i,j} \setminus c_i$ as the target; although natural, this formulation has the downside that we do not know, at fuzzing time, exactly the new coverage that can be achieved, and some of that new coverage might be far away from the old coverage; so this option would train the model with limited robustness to uncertainty; (b) use exactly one newly covered block $b \in c_{i,j} \setminus c_i$ "near" the old coverage c_i; this option is reasonable, in that we could ask a model to give us precise predictions given a single block we have not covered so far say one branch away, but has the downside of extreme cost, since each such "query" would require a model inference and there may be hundreds of nearby uncovered blocks to target; (c) use a mixture of newly covered blocks from $c_{i,j} \setminus c_i$ and of other blocks that were reachable within a small number of hops from c_i but were actually not covered as *distractors*; the benefit of this option is that it gives the model only "partial" knowledge about the exact new coverage that is achievable by the mutated argument, "local" knowledge near the old coverage that enables incremental coverage improvement within the control-flow graph, and also adds some controlled noise about other plausible reachable blocks, ensuring training robustness. We chose this third design option.

More concretely, to produce the set of targets $\hat{c_{i,j}}$, we take a "noisy" set of newly covered blocks consisting of all uncovered blocks within one branch from c_i and those in $c_{i,j} \setminus c_i$ that are only one branch away from c_i. Then we sample from this noisy set 1, 25%, 50%, 75% or 100%, ensuring there is some overlap with $c_{i,j} \setminus c_i$, and set that as our example's targets $\hat{c_{i,j}}$. To also reduce examples that have low efficiency (i.e., do not help training much), we limit the number of examples with "popular" code blocks in the target set: for every kernel code block, we count the number of examples in which it appears as a target, and we discard examples that go over a maximum. In that way, we collect our training dataset consisting of examples $\langle s_i, c_i, a_{i,j}, \hat{c_{i,j}}\rangle$.

Although machine learning can handle limited amounts of noise with sufficient dataset size, it is important to avoid systematic noise in dataset collection. One type of systematic noise in our collection process is non-determinism. As Syzkaller executes candidate mutants it uses multiple threads to dispatch system calls, to avoid getting stuck behind slow or deadlocked system calls; and it runs multiple candidates together, to avoid the cost of setting up a pristine virtual machine for every test. As a result, a test's coverage is non-deterministic. When the goal is to find a kernel crash, this non-determinism is acceptable; when a crash is detected,

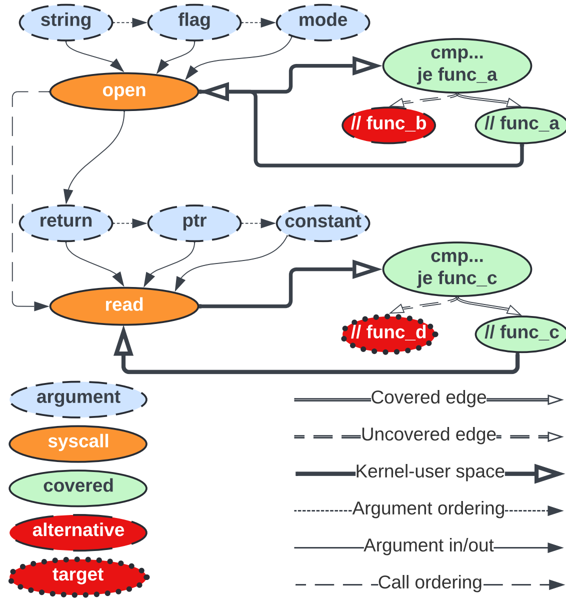


Figure 5. The representation of a mutation query for the base test of Figure 3. The test covered the green basic blocks; the red basic blocks are a single, not-taken branch away from the base test's coverage, and `func_d` is the target.

Syzkaller will attempt to reconstruct a hermetic *reproducer* that is disentangled from other tests and other sources of non-determinism, so that the bug can be investigated in a reproducible manner. However, for our purposes, this non-determinism introduces noise into our training dataset.

To reduce the noise in SNOWPLOW, first, we ensure that each test producing training examples runs from the same initial kernel state. We initialize a common execution environment, create a VM snapshot for it, and always load it before executing each test. Second, we limit concurrency by executing system calls in the test sequentially in one thread; since our goal is to generate training data (infrequently), rather than continuous fuzzing, we can afford to be occasionally stuck behind unresponsive tests. Third, SNOWPLOW avoids unnecessary interrupts to the VM. Existing fuzzers typically use the network stack to communicate with the VM (e.g., coverage collection), thus they may non-deterministically trigger guest kernel code (e.g., network code) to run [35]. SNOWPLOW replaces RPC with `virtio`, which reduces the disturbance to the guest kernel coverage.

3.2 Argument Mutation Query Representation

Developing an effective ML approach necessitates bridging the semantic gap between the mutator's inputs: the user-space program (kernel test) and the kernel coverage (both covered and desired). First, the inputs imply disparate representations. Programs are often represented as text sequences while coverage is represented in graphs—making efficient learning difficult. Second, the control dependency between the kernel test and its corresponding kernel coverage involves complex interactions among the user-space program, system calls, and kernel code. Accurately capturing these intricate dependencies is crucial for the ML model to both effectively and efficiently determine the effective mutations.

To address these challenges, our key idea is to represent the test program and coverage as a single graph, in which program and coverage are connected with explicit kernel-user space transitions. As shown in Figure 5, we transform the kernel test into a graph where nodes represent different system calls and arguments (in the `Syzlang` domain-specific language), while edges capture data and control flows in the program. Similarly, the kernel coverage is represented in a graph consisting of nodes for kernel code blocks and edges for the control flows between blocks. Finally, the test and coverage graphs are connected using edges that describe the context switches between the kernel and user space. This uniform graph representation enables ML to effectively understand the program and coverage internals while seamlessly reasoning across the user-kernel boundary.

More specifically, consider the example $\langle s_i, c_i, a_{i,j}, \hat{c_{i,j}} \rangle$, illustrated in Figure 5, s_i is the `Syzlang` base test, which we parse into an Abstract Syntax Tree. SNOWPLOW uses nodes to represent system calls and arguments in the test. For nested arguments (recall that `Syzlang` parameterizes fields of structs, and other constructor parameters for each system call), SNOWPLOW enumerates every sub-level argument according to the corresponding data structure to create argument nodes. Then, SNOWPLOW constructs edges to represent the relations among nodes, including call ordering edges that describe the control flow, argument in/out edges that capture the data flow, and argument ordering edges that show the ordering among arguments within a single system call.

For the base-test kernel coverage (c_i in our example), SNOWPLOW first converts c_i into executed kernel control flows, consisting of kernel blocks (green nodes) and control edges triggered by s_i. Then, SNOWPLOW refers to the kernel's control flow graph—obtained using static analysis—to identify code blocks reachable within one control-flow jump from c_i (red nodes), in Figure 5.

For example, SNOWPLOW detects that the uncovered `func_b` and `func_d` are reachable in a single jump each via the `cmp...` and `je` branches, so it adds the first code blocks of the two uncovered paths as alternative path entry nodes. Based on the example desirable target $\hat{c_{i,j}}$, SNOWPLOW marks the related alternative path entry nodes as the target (marked as red vertices with dotted border in the figure), informing the model of the desired mutation outcome.

Finally, SNOWPLOW connects the user-space and kernel-space inputs using kernel-user space context switch edges, which connect the system call node to the entry code block and the exit block of the system call handler. Kernel-user space context switch edges explicitly inform the model of the

control dependencies of the test and the kernel control flows. Following these edges, the model can initiate information flows that propagate between the test program tree and the kernel control flow graph and understand their relations.

To summarize, a training example consists of (1) kernel basic block vertices (containing the corresponding X86 assembly), of type "covered", "alternative" (i.e., uncovered), and "target", (2) test-program system-call vertices (containing the name of the `Syzlang` system-call variant), and (3) argument vertices (containing the type of the argument and the syscall return). Argument vertices contain information about the argument (e.g., whether it is a literal constant, a data pointer, etc.). There are control-flow edges among kernel basic blocks, edges of branches *not taken* connecting covered basic blocks to uncovered basic blocks, system-call entry and exit edges connecting system call vertices to the corresponding kernel basic blocks, and control-flow edges within the test program connecting arguments of a system call in order, inputs and outputs of a system call, and system calls in execution order. Figure 5 illustrates vertex and edge types.

3.3 *PMM* Training

Based on training examples (§3.1) and their representation as graphs (§3.2), we train the model as follows. From each example $\langle s_i, c_i, a_{i,j}, \hat{c_{i,j}} \rangle$, we construct a training sample in the form of $< x_{i,j}, y_{i,j} >$, where the input $x_{i,j}$ is the argument mutation query graph generated from the example's $s_i, c_i, \hat{c_{i,j}}$ as discussed above, and the output $y_{i,j}$ is an assignment `MUTATE` / `NOT-MUTATE` of the argument nodes inside $x_{i,j}$, so that those in $a_{i,j}$ are marked `MUTATE`.

We construct a Graph Neural Network (GNN) trained to predict vertex labels (`MUTATE` or `NOT-MUTATE` for argument vertices). Although describing the details of GNNs is out of scope in this paper, to use any GNN one must define how to *embed* the input features of the graph, i.e., how to represent the "content" of vertices and edges as vectors of floating-point numbers. We embed coverage-graph vertices as text strings of the relevant X86 assembly basic block, using a standard Transformer encoder [50]. We pre-train this encoder on all X86 assembly in a compiled Linux kernel using the BERT training recipe [36]. For program graph vertices, we treat system-call variant names and argument types as unique tokens, and use simple learned embedding matrices to embed them. We do not include literal constants (e.g., `0xffff` or `"file"`) in our representation, but use instead the type (e.g., "string" or "integer"). Edge types are similarly embedded using a learnable embedding matrix.

To sum up, Snowplow uses a model architecture with 3 learnable components: θ_{GNN} for the GNN, $\theta_{\text{TRANSFORMER}}$ for the assembly code embedding, and θ_{Emb} for embedding system call and argument nodes in the program tree and all edge types in the graph. We train the GNN by minimizing the binary cross-entropy loss between the predicted argument selection $\hat{y}_i$ and the ground truth y_i of argument nodes.

3.4 Snowplow: A Hybrid Fuzzer with *PMM*

Putting it all together, we plug *PMM* into a fuzzer (Syzkaller, in Snowplow's case) by expanding its `localizer` function (Figure 1) with *PMM*. When the fuzzer selects a base program to mutate, Snowplow creates the argument mutation query and sends it to *PMM*. Then the fuzzer starts its own mutation search, selecting a mutation type, localizing it, instantiating it, and executing it.

Because a *PMM* inference takes longer (§5.5) than a mutator iteration, we request argument localization asynchronously, and let the mutator try other types, while inference is pending. Once the localization result returns, the corresponding instantiation and execution take place.

Based on the number of predicted arguments to mutate, Snowplow dynamically adjusts the number of argument mutations for this base program. Intuitively, if a program has more arguments that are considered interesting by the *PMM*, Snowplow will mutate and execute more.

Meanwhile, the fuzzers' own mutation search will be conducted in a separate thread using existing mutation techniques. In particular, Snowplow still allows the fuzzer to perform a few random argument localizations but with a much lower probability than that of a normal fuzzer. This is mainly a fallback mechanism in case *PMM* does not catch all promising arguments and gives hand-crafted heuristic decisions by the original fuzzer a chance to help.

4 Implementation

Kernel test mutation dataset collection. Snowplow implements a test execution framework for data collection (§3.1). The framework uses the `prog` module of Syzkaller to load kernel tests in the `Syzlang` syntax and invokes the argument mutator from Syzkaller on the selected argument to generate new tests. This also included work to improve the determinism of data generation (§3.1). In total, about 7.6K LOC Golang code is written to implement the framework.

Argument mutation query representation. To create the tree representation of the test program, Snowplow uses the `prog` module of Syzkaller to parse the test, and then analyzes the control flow and data flow inside the test program by traversing the internal data structures that Syzkaller uses to describe the test. To analyze the kernel coverage, Snowplow uses Angr [43] to recover the control flow graph of the kernel and uses it to identify alternative path entry code blocks. About 2.4K Python and 1K Golang code are implemented.

Model training. *PMM* is implemented using frameworks fairseq [37] (assembly code Transformer encoder) and Pytorch Geometric framework [13] (GCN). We wrote about 0.9K LOC in Python to implement the model architecture and training, and 2.0K LOC in Python/Bash to facilitate model tuning and evaluation.

Kernel fuzzing. SNOWPLOW uses torchserve [40] to deploy *PMM*, automating model deployment on multiple GPUs, and supporting gRPC inference requests. SNOWPLOW modifies Syzkaller to invoke *PMM* for argument selection. In particular, SNOWPLOW introduces an inference worker pool, a set of goroutines responsible for creating the mutation query, invoking *PMM* for prediction, and informing Syzkaller of the selected arguments. When Syzkaller initiates the mutation of a base test, SNOWPLOW sends the test to the worker pool and immediately returns to Syzkaller, so that it can perform other mutations (e.g., call mutations) while the inference is being served. Once the inference is done, Syzkaller catches up with argument mutations using the selected arguments. Integration is implemented in 3.2K Golang LOC.

To study whether *PMM* is beneficial to other kernel fuzzing use cases, we also integrate *PMM* into SyzDirect [48], which is a state-of-the-art kernel fuzzer for directed fuzzing. The integration follows a similar design as mentioned above and takes about 2.6K LOC Golang code to implement.

5 Evaluation

We evaluate SNOWPLOW along the following questions:

RQ1: Can *PMM* localize mutations accurately? (§5.2)
RQ2: Does SNOWPLOW improve the efficiency of coverage-directed kernel fuzzing? (§5.3.1)
RQ3: Does SNOWPLOW effectively find new bugs? (§5.3.2)
RQ4: Is *PMM* beneficial to directed fuzzing? (§5.4)
RQ5: What is the performance overhead of *PMM*? (§5.5)

Experimental Setup. In total, 4 types of machines on Google Cloud Platform (GCP) are used in the evaluation of SNOWPLOW. Machine$_{collect}$ is a VM with 60 vCPUs for data collection; Machine$_{fuzz}$ is a VM with 96 vCPUs, used for fuzzing the kernels; Machine$_{train}$ is a VM with 8 A100 (40GB) GPUs, used for training; Machine$_{infer}$ is a VM with 8 L4 (24GB) GPUs, used for inference. In addition, the stable Linux kernels 6.8, 6.9, and 6.10 are used to train and evaluate SNOWPLOW.

5.1 *PMM* Training

Data collection. We build the kernel test dataset based on the publicly-accessible fuzzing artifacts of Syzbot [51], which is a project that continuously runs Syzkaller to fuzz the Linux kernel. We download the test corpus from different runs of Syzkaller and sample 1M unique kernel tests as the base test corpus. Among them, we are able to collect the kernel coverage of 0.98M tests, from which we generate the graph. Some base tests did not complete or crashed, so we excluded them from data generation. Generated training example graphs on average contain 2372 vertices (5 system call nodes, 62 argument nodes, 1631 covered code block nodes, and 674 alternative path entry nodes), and 2989 edges (39 argument ordering edges, 4 call ordering edges, 65 argument in/out edges, 1782 covered control flow edges, 1087 uncovered control flow edges, 10 kernel-user space context switch edges).

Selector	F1	Precision	Recall	Jaccard
PMModel	84.2%	91.2%	81.2%	76.1%
Rand.8	30.3%	36.6%	37.0%	19.9%

Table 1. Promising arguments selector performance. Metric scores are averaged across all base tests evaluated.

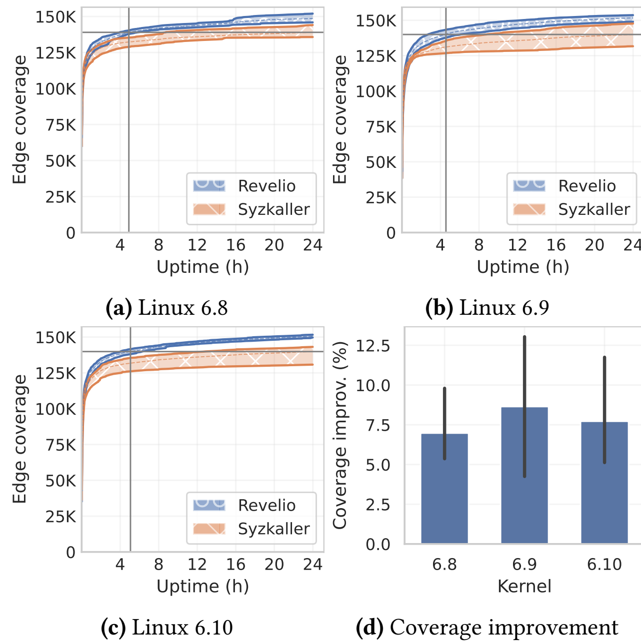
Then, we run 120 instances of machine$_{collect}$ for about 4 days to discover successful mutations—each base test is randomly mutated and executed (§3.1) 1000 times on the Linux kernel 6.8. In total, 44.3M successful mutations are discovered (about 45 per base test). We partition the examples generated from 80% of the base tests for the training dataset, and similarly 10% each for the validation and evaluation datasets, respectively. Note that a base test never appears in different dataset splits: all examples derived from a base test are either in training, or validation, or evaluation.

Hyperparameter search. We use F1 to guide the training hyperparameter search. Specifically, once a model is trained with a certain hyperparameter set, we use it to predict argument mutations for 10K base tests from the validation dataset. For every example $< x_i, y_i >$, y_i is the target set of arguments and $\hat{y}_i$ is the set of arguments selected by the trained *PMM*. Of the arguments in $\hat{y}_i$, those in y_i are true positives, while those that are not are false positives. Similarly, those in y_i but not in $\hat{y}_i$ are false negatives. Therefore, for each example, $|\hat{y}_i \cap y_i|/|\hat{y}_i|$ is the example precision and similarly $|\hat{y}_i \cap y_i|/|y_i|$ is recall. We compute per-example F1 and Jaccard similarity from these metrics, and then compute means across all examples in the validation dataset. In total, we tune the model by exploring 112 hyperparameter sets—each is trained for up to 72 hours on a machine$_{train}$, and we use the model with the highest mean F1 validation score.

5.2 *PMM* Performance

We evaluate the performance of the best trained *PMM* on 10K unique base tests from the evaluation dataset.

We compare *PMM* against a random argument localizer baseline, which randomly selects K unique arguments from each base test. We set K to 8, corresponding to the average size of y_i in the training dataset. Table 1 presents the performance of *PMM* and the baseline. First, the baseline (Rand.8) shows poor performance across all metrics, underscoring the challenge of identifying promising system call arguments. Under Jaccard Index—which measures $|\hat{y}_i \cap y_i|/|y_i \cup \hat{y}_i|$, Rand.8 averages less than 20%. In contrast, *PMM* significantly outperforms the baseline across all metrics. Its F1 score and Jaccard Index are 2.7 and 3.8 times higher than those of Rand.8, respectively, showing that *PMM* can accurately identify the promising arguments to mutate on new tests that it has not been trained for.



(a) Linux 6.8 **(b)** Linux 6.9 **(c)** Linux 6.10 **(d)** Coverage improvement

Figure 6. Edge coverage in Syzkaller and SNOWPLOW over 24 hours of fuzzing (a–c). The shaded area is bounded by the max and min coverage across 5 repeated runs. The dotted line within represents the mean. The dark horizontal line shows the mean coverage achieved by Syzkaller at 24 hours, and the dark vertical line shows the time (x-axis) when SNOWPLOW reached that same level of coverage. (d) Coverage improvement of SNOWPLOW over Syzkaller at 24 hours.

5.3 Fuzz the kernel to find new bugs

This section evaluates the fuzzing performance of SNOWPLOW on 3 recent stable Linux kernels: Linux kernel 6.8, 6.9 and 6.10; the three kernel versions were released at a 2-month cadence between March and July of 2024.

5.3.1 Edge coverage. We compare the kernel edge coverage achieved by SNOWPLOW and Syzkaller, which counts the unique edges in the triggered kernel execution path. For each kernel, we run SNOWPLOW and Syzkaller on two separate instances of $machine_{fuzz}$ to fuzz it for 24 hours, during which we collect execution traces of each fuzz test using kernel instrumentation [31]. We postprocess those execution traces (which are sequences of executed kernel basic blocks) to identify unique, directional pairs of basic blocks, or "edges". We deploy *PMM* on one instance of $machine_{infer}$, which listens to the mutation queries from SNOWPLOW over the network. We configure SNOWPLOW and Syzkaller to use the same initial seed and the same amount of VMs (42 2-vCPUs VMs) for running tests. Furthermore, the experiment is repeated 5 times, each with a unique initial seed.

Figure 6 compares edge coverage between SNOWPLOW and Syzkaller for *equal* $machine_{fuzz}$ *cost*. First, when fuzzing kernel 6.8 (Figure 6a), on which *PMM* is trained, SNOWPLOW consistently achieves higher edge coverage than Syzkaller—by 7.0% on average (Figure 6d). Moreover, the edge coverage bands of SNOWPLOW and Syzkaller do not overlap after 5 hours, meaning that across 5 runs, Syzkaller, never surpasses even the worst run of SNOWPLOW. When comparing the average edge coverage (i.e., dotted lines), SNOWPLOW requires only 4.6 hours to reach the same coverage that Syzkaller achieves in 24 hours, resulting in a 5.2x speedup. Even in its best-out-of-5 run, Syzkaller is still twice as slow as the average run by SNOWPLOW. These results demonstrate that SNOWPLOW can fuzz the kernel more efficiently.

Second, SNOWPLOW shows impressive improvement even when fuzzing the other two kernels (Figure 6b and Figure 6c), on which it was not trained. The edge coverage bands of SNOWPLOW and Syzkaller do not overlap, showing that SNOWPLOW's improvement is superior and robust. After testing for 24 hours, SNOWPLOW achieves 8.6% and 7.7% higher mean coverage than Syzkaller (Figure 6d). In terms of edge coverage speed, SNOWPLOW generally takes less than 5 hours to reach the coverage achieved by Syzkaller after 24 hours (>4.8x speed up). This shows that *PMM*, while only trained on one kernel, can generalize well to later kernels, and contribute more effective argument mutations, thereby amortizing the training cost of *PMM*.

Finally, we observe that the coverage band of Syzkaller is wider than SNOWPLOW's. SNOWPLOW achieves similar edge coverage across different initial seeds while Syzkaller's performance is sensitive to the initial seed. This is expected because most argument mutations in SNOWPLOW are guided by *PMM* while argument mutations in Syzkaller are almost random (modulo small, coverage-agnostic heuristics).

The above analysis compares the two deployments given the same $machine_{fuzz}$ resources; however SNOWPLOW has the advantage of the extra inference resources of $machine_{infer}$. We also performed a *same test-time cost* analysis (in which we give Syzkaller enough additional $machine_{fuzz}$ resources to make up for the cost of $machine_{infer}$, for various smaller instances of $machine_{infer}$ (with only CPU or fewer L4 GPUs). SNOWPLOW exhibits at least 2x speed up even in this setting. Amortizing the cost of training even in this convervative comparison is a matter of a few weeks, perhaps a drop in the ocean of continuous kernel testing.

5.3.2 Finding new bugs. We study the bug-finding ability of SNOWPLOW under an exhaustive-fuzzing setting, where SNOWPLOW and Syzkaller both fuzz the kernel for 7 days. Because the experiment is much more time-consuming, we repeat it 2 times with different seeds.

We follow a stricter set of rules than Syzbot [51] to confirm if the crash is due to a harmful kernel error. Specifically, when the guest kernel crashes, we first extract the crash description from the VM, and filter out crashes that match keywords "INFO:", "SYZFAIL", and "lost connection to the VM" because such crashes are usually less severe or too

Status	Snowplow		Syzkaller	
	run_{1st}	run_{2nd}	run_{1st}	run_{2nd}
New Crashes	67	46	0	0
Known Crashes	14	13	8	11
Total	81	59	8	11

Table 2. Crashes found during the 7-day fuzzing campaign with Snowplow and Syzkaller.

ambiguous to locate the error. Finally, we fetch the list of all kernel crashes found by Syzbot since 2018—including both fixed and unfixed bugs—and we consider a crash to be new if we cannot find the crash description in the list.

Category	Reproducer	
	Yes	No
Null pointer dereference	7	3
Paging fault	13	10
Explicit assertion violation	2	2
General protection fault	28	11
Out of bounds access	1	0
Warning	4	4
Other	2	0
Total	**57**	**30**

Table 3. New bug reports produced by Snowplow according to their manifestation. Most of the crashes have serious manifestations.

Table 2 presents the number of new and known crashes found by Snowplow and Syzkaller. First, Syzkaller does not find any new crashes, which is understandable considering Syzbot has already exhaustively tested those kernels—under different configurations—using Syzkaller. In contrast, Snowplow discovers a considerable number of new crashes. It finds 67 and 46 new crashes in the two runs and, in total, it discovers 86 crashes. Second, both Snowplow and Syzkaller discover some known crashes but Snowplow finds more. Such crashes are found in the prior runs of Syzkaller by SyzBot but they are not fixed in the kernels we test, therefore can still be triggered.

Next, we perform an additional verification on the new crashes by running the vanilla Syzkaller in bug-reproduction mode. In this mode, Syzkaller parses the execution log from the crashed VM and replays the tests to check if the crash can be triggered again. We arrive at 57 (66%) crashes for which Syzkaller's tool `syz-repro` can generate a reproducer in `Syzlang` [2]. Additionally, we found that the reproducibility rate for all crashes discovered by SyzBot is 32% (6705/20988). A common reason Syzkaller fails to reproduce certain crashes is concurrency [4], which is inherently difficult to replicate. We further run `syz-symbolize` [25], which parses the kernel console log, to locate the kernel code involved in each crash.

Table 3 categorizes the new crashes based on the patterns of the crash description. It shows that most crashes, with or without a reproducer, can have serious impact on the kernel by causing kernel errors and even security issues, such as out-of-bounds reads (detected by KASAN) in the kernel. While SyzBot automatically reports all crashes to kernel developers, we opted for a more developer-friendly approach, aligning with expectations for research prototypes. Specifically, we manually analyzed each crash before reaching out to developers, ensuring we had preliminary findings to share. Although this approach is more time-consuming, it fosters better collaboration with the developer community. In total, we diagnosed 7 unique crashes (out of our 57 reproducible crashes) over approximately 50 hours and reported them to developers, as presented in Table 4. As of this writing, 4 bugs have been confirmed, with 2 already fixed.

In particular, Snowplow discovered an out-of-bounds write bug [7] in the Linux kernel ATA driver, which allowed the driver thread to overwrite arbitrary kernel memory pages with data read from the device, leading to kernel panics or potential security vulnerabilities. Despite the driver being actively maintained, the bug had remained unnoticed in the kernel for nearly two decades before Snowplow identified it. Our analysis shows that this bug is difficult to find because the problematic code is only executed when the system call request is configured with carefully crafted parameters. Specifically, a kernel test must invoke the `ioctl()` system call on a SCSI device file descriptor, setting the command request to `SCSI_IOCTL_SEND_COMMAND`. Meanwhile, the call arguments should set the SCSI command to `ATA_16 PASS-THROUGH` mode, the ATA command to `ATA_NOP`, and the protocol to `ATA_PROT_PIO`. Finally, an incorrect data length in the call argument triggers the out-of-bounds write due to an insufficient boundary check in the corresponding function.

Due to the strict constraints on system call arguments, this bug is particularly difficult to uncover using Syzkaller's random mutations, which struggle to find the precise argument combinations needed to reach the problematic code path. In contrast, Snowplow leverages *PMM* to systematically identify promising arguments of `ioctl` that trigger new kernel execution paths. By guiding Syzkaller to keep mutating these critical arguments as long as the model predicts new resulting coverage, Snowplow effectively triggers the bug, ultimately leading to its discovery.

Since the out-of-bounds write bug can corrupt arbitrary kernel memory, leading to kernel crashes at different locations, we investigated whether other crashes might stem from the same issue. We analyzed all crash reproducers and found that 45 out of 57 contained the `ioctl()` system call with the command `SCSI_IOCTL_SEND_COMMAND`. Given this strong correlation, we conservatively attribute these 45 crashes to

[2] Crash reports and reproducers are available in the Github repository.

ID	Bug description	Detector	Failure context / syscall	Failure location	Status
1	Out of bound access in ata_pio_sector	N/A	ioctl()	drivers/ata/	Fixed
2	General Protection Fault in native_tss_update_io_bitmap	N/A	io_uring()	arch/x86/kernel/	Fixed
3	RCU stall in __sanitizer_cov_trace_pc	RCU stall detector	Timer interrupt	kernel/	Confirmed
4	GUP (Get User Pages) no longer grows the stack	Built-in checker	mmap()	mm/	Confirmed
5	WARNING in ext4_iomap_begin	WARN_ON()	pwrite64()	fs/ext4/	Reported
6	kernel BUG in ext4_do_writepages	BUG()	Filesystem background operation	fs/ext4/	Reported
7	KASAN: slab-use-after-free Read in ext4_search_dir	KASAN	open()	fs/ext4/	Reported

Table 4. Sample of 57 reproducible bugs reported by Snowplow. Bugs #1-#7 were reported to developers, and bugs #1-#4 have already been confirmed and/or fixed.

the discovered bug, while the remaining 12 crashes (6 already diagnosed and reported) likely result from other issues.

5.4 Kernel Directed Fuzzing

Directed fuzzing has recently emerged as a promising technique to reproduce bugs and perform regression testing. Unlike traditional fuzzers that aim at the highest kernel coverage, directed fuzzers try to reach one or more target code region(s) specified by the user. But similar to normal fuzzers, directed fuzzers still heavily rely on test mutation to create new tests. This section evaluates the effectiveness of *PMM* at accelerating directed kernel fuzzing.

SyzDirect—the state-of-the-art directed kernel fuzzer—performs mutations in a manner similar to Syzkaller, but prioritizes mutations using a number of sophisticated heuristics (e.g., choosing system calls that allocate resources needed by the code at the target, choosing arguments in earlier system calls that will enable issuing the right system call later in the test, choosing test to mutate by selecting the test that achieved coverage closest to the target, etc.). As with Syzkaller, we use *PMM* to select arguments when SyzDirect's heuristics choose to mutate arguments, and SyzDirect's own heuristics for remaining mutations. We call the resulting hybrid SyzDirect with *PMM Snowplow-D*.

We compare SyzDirect and Snowplow-D using the same test dataset on which SyzDirect is originally evaluated. The dataset [18] consists of 100 unique kernel code locations that are related to bugs found by SyzBot. Because the dataset does not provide the exact kernel versions—and the original dataset applies them to many development kernel versions—we look up the bug list in SyzBot to find the kernel commits that are associated with the bug documentation and randomly select one commit to use. Following SyzDirect's instructions, which involve applying a customized kernel patch and compiling with a custom LLVM compiler, we are able to run SyzDirect on 24 out of those 100 code locations. For the remaining 76 code locations, we could not automatically apply the kernel patch required by SyzDirect [19], and the project offered no instructions on resolving such conflicts.

We evaluate the time taken for Snowplow-D and SyzDirect to reach (i.e., cover) the target code location. We focus on *reaching* the target code instead of causing a crash, since our goal is to measure the ability of Snowplow-D to speed up directed fuzzing, and because the technique can be used also to validate a patch (rather than causing crashes). For each target code location, we run one machine$_{fuzz}$ for SyzDirect and one for Snowplow-D for up to 24 hours, and repeat the measurement 5 times to compute the average time taken for the search. As shown in Table 5, SyzDirect reaches 19 code locations while Snowplow-D reaches 2 more locations (21 in total). In particular, a few code regions are particularly hard to reach while the remaining ones can usually be reached in less than 3 minutes. Our inspection shows that the easy-to-reach code regions are typically located at the entry point of a system call and can be reached easily as long as the right system call is invoked. Thus, it is expected that Snowplow-D achieves similar or worse—due to the inference overhead—performance as SyzDirect because *PMM* focuses on the argument mutations.

Our analysis showed that the hard-to-reach regions reside on the deeper branches of the system-call handler, which naturally can benefit from more effective argument mutations. This is consistent with the observed improvement of Snowplow-D. In total, Snowplow-D is 8.5x faster than SyzDirect in reaching the 19 code locations that both systems can reach. The significant improvement on the hard-to-reach regions suggests a hybrid design where SyzDirect performs the initial exploration using its heuristics and resorts to *PMM* if the target is challenging to reach.

Interestingly, all the evaluated kernels were released 2–3 years ago, but *PMM* can still predict effective mutations that accelerate the directed exploration, underscoring the generalization ability of *PMM* to various kernel versions.

5.5 Performance characteristics

Inference performance. We deploy the trained *PMM* on one instance of machine$_{infer}$ and then launch various concurrent fuzzing clients to understand inference performance. At saturation, the machine serves about 57 queries per second, and the average latency for a query is 0.69 seconds.

Fuzzing performance. Snowplow achieves similar fuzzing throughput as Syzkaller. When fuzzing kernel 6.10 on one instance of machine$_{fuzz}$, the throughput of Snowplow is 383 tests per second, and 390 for Syzkaller. This is because the

Target Location	Kernel	SyzDirect	Snowplow-D	Speedup
net/rxrpc/sendmsg.c:747	438645193e59	NA (0/5)	41 (1/5)	INF
drivers/infiniband/core/cma.c:2584	555f3d7be91a	NA (0/5)	4376 (1/5)	INF
drivers/scsi/scsi_ioctl.c:357	debe436e77c7	24487 (1/5)	101 (1/5)	242.4
kernel/watch_queue.c:205	f443e374ae13	36093 (1/5)	1004 (1/5)	35.9
net/netfilter/nf_tables_api.c:7233	169387e2aa29	21083 (1/5)	5158 (2/5)	4.1
net/mac80211/iface.c:2008	1286cc4893cf	10848 (2/5)	4011 (2/5)	2.7
xdp/xdp_umem.c:101	f4bc5bbb5fef	48 (4/5)	18 (4/5)	2.7
net/packet/af_packet.c:4461	a763d5a5abd6	66 (3/5)	41 (2/5)	1.6
net/core/dev.c:3701	2585cf9dfaad	65 (4/5)	46 (5/5)	1.4
fs/io_uring.c:8999	7f25f0412c9e	70 (1/5)	57 (1/5)	1.2
fs/ext4/extents_status.c:897	14702b3b2438	110 (3/5)	101 (2/5)	1.1
fs/f2fs/node.c:611	09688c0166e7	25 (4/5)	23 (4/5)	1.1
sound/core/oss/pcm_plugin.c:70	68453767131a	42 (5/5)	42 (5/5)	1.0
net/bluetooth/sco.c:523	f443e374ae13	43 (4/5)	44 (5/5)	1.0
fs/btrfs/volumes.c:1361	2293be58d6a1	65 (5/5)	68 (5/5)	1.0
fs/ext4/super.c:6607	cd8c917a56f2	25 (5/5)	31 (3/5)	0.8
mm/madvise.c:213	73878e5eb1bd	30 (2/5)	42 (3/5)	0.7
kernel/dma/mapping.c:263	aad611a868d1	17 (5/5)	24 (5/5)	0.7
drivers/video/fbdev/core/fbcon.c:2436	fa55b7dcdc43	18 (3/5)	26 (3/5)	0.7
fs/aio.c:2001	6f513529296f	19 (5/5)	34 (5/5)	0.6
fs/erofs/decompressor.c:227	a51e3ac43ddb	41 (5/5)	101 (5/5)	0.4
Subtotal		93195	10972	8.5
kernel/watch_queue.c:273	34e047aa16c0	NA (0/5)	NA (0/5)	NA
kernel/cgroup/cgroup.c:3629	1d1df41c5a33	NA (0/5)	NA (0/5)	NA
crypto/crypto_null.c:88	5859a2b19911	NA (0/5)	NA (0/5)	NA

Table 5. Average time to reach the bug-related kernel code in seconds and success rate (successful/total runs). Subtotal shows the time taken to reproduce bugs that both systems reproduced.

.

inference is handled by concurrent goroutines and does not block any critical paths (§4).

6 Discussion and Limitations

Other types of mutations. Although our results show that our intervention produces a powerful hybrid fuzzer, we only demonstrate the methodology for localization in argument mutation. We believe that the *PMM* modelling approach will readily generalize to a number of other mutation types, both for localization and instantiation. For example, the methodology can be used to localize system call insertion with no representational or training changes. We could further enhance our graphs trivially to enable an instantiation prediction as well (choose one of the known system-call variants for the insertion). This would require predicting not a binary `MUTATE/NOT-MUTATE` label for the localized system-call node, but instead one of the thousand or so system-call mutants, which is a minimal change in the architecture. Capturing other mutation types would work in a similar fashion, and we expect that training a single model for multiple mutation types will, in fact, *improve* performance, since the learning tasks are very similar and subject to very positive knowledge transfer. One mutation type that would not work well with Snowplow is splicing different tests together (similar to genetic mutation algorithms). That "non-local" style of mutation would require a different approach.

Return on Investment. Snowplow makes a bet that the cost of obtaining training examples, and training *PMM*, are together significantly lower than the benefit of reaching given coverage faster. This bet pays off thanks to model generalization: *PMM* achieves the coverage that Syzkaller does in 24 hours 5 times faster for multiple kernel versions over four months. Approximately, after 8.75 days of continuous fuzzing Snowplow breaks even with Syzkaller. This is oversimplified, since Snowplow also needs relatively expensive GPUs to train for 3 days, and inexpensive GPUs to serve the model. Thankfully, given that the kernel-developer community performs continuous fuzzing of the kernel *for years(!)*, training *PMM* for a few days every once in a while (and potentially sharing the model weights among different institutions for their own uses, e.g., proprietary versions of the kernel) is well amortized many times over. However, for one-off fuzzing compaigns of vastly different kernel architectures, the cost of training *PMM* might not pay off.

Coverage Dynamics. Our evaluation looks at "early" coverage, starting from scratch and getting to a coverage plateau.

Although directed fuzzing approximates "late" coverage behavior where a singular block remains uncovered, there are no good examples of hard-to-reach, persistent uncovered blocks. Thankfully, Snowplow is designed to be targeted. In future work, we will run localized saturation campaigns (within a single driver or subsystem) so we can more exhaustively understand *PMM* performance in saturated settings.

7 Related work

Kernel test mutation. Moonshine [38] and Healer [46] improve insert-new-call mutations by discovering implicit control dependencies between system calls and then encouraging the discovered dependencies during mutations. Because the kernel is highly stateful, the execution of one system call A may affect the execution of the following call B, constituting a control dependency of call B on call A. To find the dependencies, Moonshine profiles system call sequences in real-world applications. Healer proposes a statistical method to infer implicit system call dependencies from corpus tests that trigger unique kernel coverage.

HFL [32] uses symbolic execution to guide argument mutations. To achieve this, it introduces several kernel modifications to enable an existing symbolic solver [10] to work on low-level kernel code. However, due to the high cost of symbolic analysis, HFL employs the symbolic engine as a fallback to the random argument mutations. Specifically, HFL only performs expensive symbolic analysis when a code branch remains uncovered after a certain number of mutations. In contrast, Snowplow uses *PMM* as a first-choice mutator for every base test, thanks to *PMM*'s high efficiency (§5.5).

ML in kernel fuzzing. SyzVegas [52] uses reinforcement learning to schedule high-level fuzzing tasks such as when to generate new programs from scratch or through mutations, which otherwise is typically decided by the hard-coded weights. To achieve this goal, SyzVegas proposes a reward model that estimates the priority of different fuzzing tasks. However, SyzVegas still relies on randomness to control the lower-level fuzzing tasks such as test mutation. As such, it is orthogonal to Snowplow, which focuses on test mutation.

In Snowcat [22] we proposed an ML model to improve test selection in kernel concurrency testing [16]. It trains a model that can predict kernel code coverage of a multi-threaded kernel test and uses that to select promising tests to execute. Compared to it, Snowplow predicts mutations of the test given desirable coverage as input, rather than predicting coverage given a thread interleaving. It would be interesting future work to treat interleaving as an additional mutation type in the context of Snowplow, or combine it with a coverage predictor similar to that of Snowcat.

KernelGPT [54] leverages large language models to automatically generate system call specifications, which are essential for kernel fuzzers to produce valid tests (e.g., correct argument types) but typically require kernel experts to write manually [45]. Snowplow complements KernelGPT by focusing on mutating generated base tests. Together, these approaches enhance different stages of the fuzzing pipeline—KernelGPT facilitates test generation, while Snowplow improves test mutation.

ML in software optimization. SmartChoices [6] introduces a learning framework and new programming APIs that allow developers to easily apply machine learning algorithms to control important decisions in the program. In the same vein, PSS [56] proposes a prediction service that can interact with the caller to provide predictions and incorporate the feedback. Lake [14] explores how to efficiently and securely use machine learning in the kernel-space so that heuristics in the kernel can be replaced with machine learning algorithms. Compared to them, Snowplow explores a dedicated ML approach for kernel test mutations, which can learn to predict the outcome (e.g., argument localization) from a sequence of decisions—heuristics and randomnesses in the existing argument selection—all at once.

8 Conclusion

This work proposes Snowplow, a kernel fuzzer that uses a learned white-box test mutation strategy to perform effective test mutations. Snowplow builds the mutator using a new machine learning model called *PMM*, which inputs the base test, its kernel coverage, and the desired target coverage and outputs promising mutations to reach the desired coverage. Snowplow is implemented to learn the argument mutations as a case study and evaluated on recent stable Linux kernel releases, showing that learned mutation guidance can significantly speed up coverage increase, help find new bugs in the kernel, and considerably accelerate directed kernel fuzzing. We are excited to explore the impact of this powerful and practical new technique in making software safe and secure.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Qirun Zhang, for their insightful feedback. We also thank the Reliable and Secure System Lab members for their detailed and helpful comments on this work and earlier drafts. Finally, we appreciate the precise and constructive suggestions by the kernel dynamic analysis team at Google, and in particular Dmitry Vyukov and Aleksandr Nogikh. This work was funded in part by the National Science Foundation (NSF) under grants CNS-2140305 and CNS-2145888 and gifts by Google and Intel.

References

[1] Muhammad Abubakar, Adil Ahmad, Pedro Fonseca, and Dongyan Xu. 2021. SHARD: Fine-Grained Kernel Specialization with Context-Aware Hardening. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2435–2452. https://www.usenix.org/conference/usenixsecurity21/presentation/abubakar

[2] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2329–2344. doi:10.1145/3133956.3134020

[3] Ella Bounimova, Patrice Godefroid, and David Molnar. 2013. Billions and billions of constraints: whitebox fuzz testing in production. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 122–131.

[4] Joseph Bursey, Ardalan Amiri Sani, and Zhiyun Qian. 2024. SyzRetrospector: A Large-Scale Retrospective Study of Syzbot. arXiv:2401.11642 [cs.SE] https://arxiv.org/abs/2401.11642

[5] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[6] Victor Carbune, Thierry Coppey, Alexander Daryin, Thomas Deselaers, Nikhil Sarda, and Jay Yagnik. 2019. SmartChoices: Hybridizing Programming and Machine Learning. In *Reinforcement Learning for Real Life (RL4RealLife) Workshop in the 36th International Conference on Machine Learning (ICML)*,. https://arxiv.org/abs/1810.00619

[7] Niklas Cassel. Online. [PATCH] ata: libata-sff: Ensure that we cannot write outside the allocated buffer. https://lore.kernel.org/all/173806124194.7504.9074817431897226887.b4-ty@kernel.org/T/.

[8] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-Adaptive Mutational Fuzzing. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, USA, 725–741. doi:10.1109/SP.2015.50

[9] Weiteng Chen, Yu Hao, Zheng Zhang, Xiaochen Zou, Dhilung Kirat, Shachee Mishra, Douglas Schales, Jiyong Jang, and Zhiyun Qian. 2024. SyzGen++: Dependency Inference for Augmenting Kernel Driver Fuzzing. In *IEEE Symposium on Security and Privacy*.

[10] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2012. The S2E Platform: Design, Implementation, and Applications. *ACM Trans. Comput. Syst.* 30, 1, Article 2 (Feb. 2012), 49 pages. doi:10.1145/2110356.2110358

[11] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. 2019. Grey-box concolic testing on binary code. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 736–747. doi:10.1109/ICSE.2019.00082

[12] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability modulo theories: introduction and applications. *Commun. ACM* 54, 9 (Sept. 2011), 69–77. doi:10.1145/1995376.1995394

[13] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[14] Henrique Fingler, Isha Tarte, Hangchen Yu, Ariel Szekely, Bodun Hu, Aditya Akella, and Christopher J. Rossbach. 2023. Towards a Machine Learning-Assisted Kernel with LAKE. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 846–861. doi:10.1145/3575693.3575697

[15] Marius Fleischer, Dipanjan Das, Priyanka Bose, Weiheng Bai, Kangjie Lu, Mathias Payer, Christopher Kruegel, and Giovanni Vigna. 2023. ACTOR: Action-Guided Kernel Fuzzing. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 5003–5020. https://www.usenix.org/conference/usenixsecurity23/presentation/fleischer

[16] Pedro Fonseca, Rodrigo Rodrigues, and Björn B. Brandenburg. 2014. SKI: exposing kernel concurrency bugs through systematic schedule exploration. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 415–431.

[17] Pedro Fonseca, Xi Wang, and Arvind Krishnamurthy. 2018. MultiNyx: a multi-level abstraction framework for systematic analysis of hypervisors. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 23, 12 pages. doi:10.1145/3190508.3190529

[18] SecLab Fudan. 2023. SyzDirect evaluation dataset. https://github.com/seclab-fudan/SyzDirect/blob/main/source/syzdirect/Runner/dataset.xlsx.

[19] SecLab Fudan. 2023. SyzDirect kernel modification. https://raw.githubusercontent.com/seclab-fudan/SyzDirect/refs/heads/main/source/syzdirect/kcov.diff.

[20] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Commun. ACM* 55, 3 (March 2012), 40–44. doi:10.1145/2093548.2093564

[21] Sishuai Gong, Deniz Altinbüken, Pedro Fonseca, and Petros Maniatis. 2021. Snowboard: Finding Kernel Concurrency Bugs through Systematic Inter-thread Communication Analysis. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 66–83. doi:10.1145/3477132.3483549

[22] Sishuai Gong, Dinglan Peng, Deniz Altınbüken, Pedro Fonseca, and Petros Maniatis. 2023. Snowcat: Efficient Kernel Concurrency Testing using a Learned Coverage Predictor. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 35–51. doi:10.1145/3600006.3613148

[23] Google. Online. Filesystem related system call variants in Syzkaller. https://github.com/google/syzkaller/blob/master/sys/linux/filesystem.txt.

[24] Google. Online. Syscall description language. https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions_syntax.md.

[25] Google. Online. syz-symbolize. https://github.com/google/syzkaller/blob/master/tools/syz-symbolize/symbolize.go.

[26] Google. Online. Syzkaller-kernel fuzzer. https://github.com/google/syzkaller.

[27] Yu Hao, Hang Zhang, Guoren Li, Xingyun Du, Zhiyun Qian, and Ardalan Amiri Sani. 2022. Demystifying the dependency challenge in kernel fuzzing. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 659–671. doi:10.1145/3510003.3510126

[28] Intel. Online. HW-assisted Feedback Fuzzer for x86 VMs. https://github.com/IntelLabs/kAFL.

[29] Ling Jiang, Hengchen Yuan, Mingyuan Wu, Lingming Zhang, and Yuqun Zhang. 2023. Evaluating and Improving Hybrid Fuzzing. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 410–422. doi:10.1109/ICSE48619.2023.00045

[30] Dave Jones. Online. Trinity: Linux system call fuzzer. https://github.com/kernelslacker/trinity.

[31] The kernel development community. Online. KCOV: code coverage for fuzzing. https://docs.kernel.org/dev-tools/kcov.html.

[32] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/hfl-hybrid-fuzzing-on-the-linux-kernel/

[33] Christopher Kruegel, Darren Mutz, Fredrik Valeur, and Giovanni Vigna. 2003. On the Detection of Anomalous System Call Arguments. In *Computer Security – ESORICS 2003*, Einar Snekkenes and Dieter Gollmann (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 326–343.

[34] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings*

of the 33rd ACM/IEEE International Conference on Automated Software Engineering (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 475–485. doi:10.1145/3238147.3238176

[35] Congyu Liu, Sishuai Gong, and Pedro Fonseca. 2023. KIT: Testing OS-Level Virtualization for Functional Interference Bugs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 427–441. doi:10.1145/3575693.3575731

[36] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. arXiv:1907.11692 [cs.CL]

[37] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. 2019. fairseq: A Fast, Extensible Toolkit for Sequence Modeling. In *Proceedings of NAACL-HLT 2019: Demonstrations*.

[38] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 729–743. https://www.usenix.org/conference/usenixsecurity18/presentation/pailoor

[39] Sebastian Poeplau and Aurélien Francillon. 2021. SymQEMU: Compilation-based symbolic execution for binaries. In *Network and Distributed System Security Symposium*. Network & Distributed System Security Symposium.

[40] pytorch. Online. TorchServe. https://github.com/pytorch/serve.

[41] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: hardware-assisted feedback fuzzing for OS kernels. In *Proceedings of the 26th USENIX Conference on Security Symposium* (Vancouver, BC, Canada) *(SEC'17)*. USENIX Association, USA, 167–182.

[42] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2019. NEUZZ: Efficient Fuzzing with Neural Program Smoothing. In *2019 IEEE Symposium on Security and Privacy (SP)*. 803–817. doi:10.1109/SP.2019.00052

[43] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. 138–157. doi:10.1109/SP.2016.17

[44] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution.. In *NDSS*, Vol. 16. 1–16.

[45] Hao Sun, Yuheng Shen, Jianzhong Liu, Yiru Xu, and Yu Jiang. 2022. KSG: Augmenting Kernel Fuzzing with System Call Specification Generation. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 351–366. https://www.usenix.org/conference/atc22/presentation/sun

[46] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. 2021. HEALER: Relation Learning Guided Kernel Fuzzing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 344–358. doi:10.1145/3477132.3483547

[47] SyzBot. Online. KASAN: slab-use-after-free Read in unix_stream_read_actor (2). https://syzkaller.appspot.com/bug?extid=8811381d455e3e9ec788.

[48] Xin Tan, Yuan Zhang, Jiadong Lu, Xin Xiong, Zhuang Liu, and Min Yang. 2023. SyzDirect: Directed Greybox Fuzzing for Linux Kernel. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 1630–1644. doi:10.1145/3576915.3623146

[49] Shobha Vasudevan, Wenjie (Joe) Jiang, David Bieber, Rishabh Singh, hamid shojaei, C. Richard Ho, and Charles Sutton. 2021. Learning Semantic Representations to Verify Hardware Designs. In *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (Eds.), Vol. 34. Curran Associates, Inc., 23491–23504. https://proceedings.neurips.cc/paper/2021/file/c5aa65949d20f6b20e1a922c13d974e7-Paper.pdf

[50] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[51] Dmitry Vyukov. 2018. Introducing the syzbot dashboard. https://lwn.net/Articles/749910/.

[52] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V. Krishnamurthy, and Nael Abu-Ghazaleh. 2021. SyzVegas: Beating Kernel Fuzzing Odds with Reinforcement Learning. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2741–2758. https://www.usenix.org/conference/usenixsecurity21/presentation/wang-daimeng

[53] Jiacheng Xu, Xuhong Zhang, Shouling Ji, Yuan Tian, Binbin Zhao, Qinying Wang, Peng Cheng, and Jiming Chen. [n. d.]. MOCK: Optimizing Kernel Fuzzing Mutation with Context-aware Dependency. ([n. d.]).

[54] Chenyuan Yang, Zijie Zhao, and Lingming Zhang. 2023. KernelGPT: Enhanced Kernel Fuzzing via Large Language Models. https://arxiv.org/abs/2401.00563. arXiv:2401.00563 [cs.CR]

[55] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 745–761. https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[56] Zhizhou Zhang, Alvin Oliver Glova, Timothy Sherwood, and Jonathan Balkind. 2023. A Prediction System Service. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 48–60. doi:10.1145/3575693.3575714