Leveraging Binary Coverage for Effective Generation Guidance in Kernel Fuzzing

Jianzhong Liu
Tsinghua University
Beijing, China
liujz21@mails.tsinghua.edu.cn

Yuheng Shen
Tsinghua University
Beijing, China
shenyh20@mails.tsinghua.edu.cn

Yiru Xu
Tsinghua University
Beijing, China
xuyr21@mails.tsinghua.edu.cn

Yu Jiang
Tsinghua University
Beijing, China
jiangyu198964@126.com

Abstract

State-of-the-art kernel fuzzers use *edge-based* code coverage metrics for novel behavior detection. However, code coverage is not sufficient for operating system kernels, for they contain many untracked but interesting features, such as comparison operands, kernel state identifiers, flags, and executable code, within its data segments, that reflects different execution patterns, and can profoundly increase the granularity and scope of the coverage metrics.

This paper proposes the use of *Kernel Binary Coverage Feedback*, a comprehensive and effective execution feedback method that provides metrics reflecting the execution coverage status of the entire binary coverage to kernel fuzzers. Our approach abstracts program behavior as its memory access pattern during execution, and considers all such relevant behavior, including standard memory reads and writes, predicate comparisons, etc., to obtain a coverage metric on the whole kernel binary for input generation guidance.

We implemented a prototype tool KBinCov and integrated it into a popular kernel fuzzer Syzkaller. We evaluated its effectiveness against vanilla Syzkaller, as well as certain other approaches, including StateFuzz and IJON. Our results show that KBinCov achieves code and binary coverage increases of 7%, 7%, 9%, and 87%, 34%, 61%, compared to Syzkaller (using kcov), StateFuzz, and IJON, on recent versions of the Linux kernels, respectively, while only incurring a 1.74× overhead increase, *less* than StateFuzz and IJON's 2.5× and 2.2× figures. In addition, we found 21 previously unknown bugs using KBinCov with Syzkaller, more than Syzkaller (with kcov), StateFuzz, and IJON, which found 4, 4, and 2 bugs, respectively.

CCS Concepts

• **Security and privacy → Operating systems security**; **Vulnerability management**;

Keywords

kernel fuzzing, coverage-guided fuzzing, operating systems security, software testing

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690232

ACM Reference Format:
Jianzhong Liu, Yuheng Shen, Yiru Xu, and Yu Jiang. 2024. Leveraging Binary Coverage for Effective Generation Guidance in Kernel Fuzzing. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690232

1 Introduction

Kernel fuzzing is a popular and effective technique for finding bugs in operating system kernels. It feeds large quantities of generated input to the program-under-test, with operating system kernels as its targets. To perceive the execution status of the program-under-test and facilitate effective fuzzing input generation, code coverage is commonly utilized as feedback guidance for execution behavior novelty detection. State-of-the-art kernel fuzzers, such as Syzkaller [27], utilize this technique to keep the *interesting* inputs, i.e., those that trigger new kernel execution behavior, for further analysis and generation.

However, current kernel fuzzing technologies can only allow fuzzers to explore a less-than-desired amount of the kernel's code and logic, typically reaching their limits within several days into a fuzzing campaign, where it reaches only *shallow* code, i.e., those that can be visited by satisfying few constraints, of the kernel's logic. This is due to operating system kernels containing very complex logic and an unfathomable number of states, where basic code coverage can only identify the novelty of a small the proportion of these execution behaviors, leading to a significantly reduced capacity in input generation, resulting in reduced fuzzing effectiveness and an inability to detect more intricate bugs efficiently.

There have been some prior research efforts to mitigate such a deficiency. Initially, fuzzers such as AFL++ [9] incorporated context, i.e. where the state of the program was when the code was executed, into coverage metrics, thus allowing for slightly higher effectiveness. However, this is still restricted to covering the code section alone. Another approach pioneered by Aschermann et al. is IJON [1], which uses manually specified *state* variables and their values to complement code coverage in detecting novel state changes. While IJON is applicable for smaller projects, the operating system kernel's codebase is too large and complicated for any feasible application of manual state variable identification, thus rendering IJON insufficient. StateFuzz [32] is another approach that targets

kernel drivers directly, using manually specified criteria to automatically filter variables that reflect the kernel's runtime state and deduce value ranges for its states. While automated deduction is more applicable, its design cannot be scaled across the entire kernel.

Our observation is that the execution behavior of an operating system kernel can be abstracted into its temporal and spatial memory access patterns into the entire region mapped by the kernel binary. In this sense, traditional code coverage can only cover a subset of such behavior, identifying only the memory access patterns for control flow features in the code sections, mainly in the form of accessing a new code block when performing branch, function call, and return instructions. Memory accesses to data-relevant features in the kernel binary are indicative of run-time behavior, such as feature and status flags, comparison operands, global objects, etc., allowing the fuzzer to further perceive novel execution traces, thus guiding the fuzzer towards generating more effective inputs, consequently exploring more states and discovering more bugs.

In this paper, we propose the use of *Kernel Binary Coverage Feedback* for kernel fuzzers. In essence, this method traces not only control flow features during the kernel's execution, such as unconditional jumps, conditional branches, function entries and exits, etc., but also memory accesses static values within the binary itself, consisting of immediate values, global values, etc. While the concept is straightforward, effectively increasing a kernel fuzzer's feedback efficacy requires both *efficiency* and *accuracy*, which are often mutually contradictory and require making design compromises. For instance, a naïve implementation of coverage storage is a bit vector with a length of nearly 1.1×10^{12} bytes[1], an intractable amount for efficient executions.

For this method to be feasible in kernel fuzzing, we make the following design choices to achieve preferable accuracy and efficiency. *First, for data embedded in instructions,* we use address approximation to reduce the complexity required for calculating its specific address without sacrificing accuracy. *Next, for situations where the instruction semantics differ from the syntax,* we perform specific length calculations, and allow manually-specified, platform-dependent calculations extensions for increased precision. *Then, using static analysis during instrumentation, we perform tracing elision,* an optimization that removes tracing points that contribute redundant coverage information. *Finally, for efficient coverage storage,* we propose multi-level cache-friendly coverage storage that preserves accuracy while ensuring efficiency.

We implement a prototype feedback tool KBinCov that uses *Kernel Binary Coverage Feedback* for kernel fuzzers, and integrated it with a state-of-the-art kernel fuzzer Syzkaller. To understand the effectiveness of our design, we evaluate KBinCov's effectiveness compared to vanilla Syzkaller's code coverage feedback, and StateFuzz's and IJON's state variable coverage feedback, where the latter records the state of variables deduced by the former. Our results show that, KBinCov assists Syzkaller in achieving a 7%, 7%, and 9% code and 87%, 34%, 61% binary coverage improvement compared to vanilla Syzkaller, StateFuzz, and IJON, respectively. Furthermore, our findings show that compared to vanilla Syzkaller, KBinCov incurs an overhead of 1.74×, which is *more efficient* than the state coverage approaches. Ablation tests show that reversing our design choices or restricting collection metrics incurs overheads (up to 50%), or reduces the capabilities of KBinCov's feedback mechanisms (the original version leads 7% to 30%). Comparing binary coverage and code coverage also shows that our approach encompasses code coverage while providing more state information regarding the kernel's execution. During the fuzzing campaign, KBinCov found 21 previously unknown bugs, whereas Syzkaller (with kcov), StateFuzz, and IJON found 4, 4, and 2 bugs.

Our contributions in this paper are mainly as follows.

- We propose *Kernel Binary Coverage Feedback* as a new execution feedback and novelty detection mechanism for kernel fuzzers to increase their perception into the kernel's execution behavior.
- We implement KBinCov, a prototype tool that uses binary coverage to record the relevant access patterns and detect novel execution behavior for Syzkaller.
- We evaluate how KBinCov assists Syzkaller in its fuzzing effectiveness, where our results show that binary coverage can indeed improve the effectiveness of fuzzing input generation, through increased coverage statistics and bugs found in contrast to previous attempts, while only incurring a reasonable overhead as a compromise.

2 Background

2.1 Kernel Fuzzing

Fuzzing is a popular program testing technique for detecting concrete bugs within software systems with an emphasis on soundness and scalability. It tests targets by repeatedly feeding randomly or partially-randomly generated input payloads to the program or system under test to explore its code space and attempt to trigger bugs within. Its testings targets range from command-line utilities (e.g. *binutils*), to large and composite software systems, such as databases, protocol stack implementations, machine learning frameworks, and operating system kernels. Fuzzing has achieved great success in finding various bugs in many real-world software systems, and has attracted research attention from both academia and industry. While the inputs of these targets vary, many techniques remain similar, including feedback-based greybox fuzzing, which uses execution feedback as guidance for further input generation.

Many researchers have attempted to utilize fuzzing for finding kernel bugs, thus improving the kernel's overall security. The current state-of-the-art in kernel fuzzing is Google's Syzkaller, which is implemented by mainly following the common design paradigms of userspace program fuzzers. Generally speaking, fuzzing a kernel involves the following steps: 1) the fuzzer runs the target kernel within a virtualized or emulated environment; 2) it feeds the target kernel with generated test cases, usually consisting of system calls sequences; 3) the fuzzer then leverages kernel feedback information such as coverage to find bugs and guide further input generation; 4) it also monitors for any exceptional behavior and reports any crashes found, typically with kernel sanitizers [10, 11].

Syzkaller mainly utilizes code coverage feedback obtained from the kcov utility, commonly found in many operating system kernels. The coverage metric of a kernel is different than that of a userspace program as kernels are concurrent programs that may run multiple

[1]Contemporary 64-bit kernels generally use effective addresses 48-bits or 57-bits wide, occupying a bit vector of at least 2^{48} bits $\approx 1.1 \times 10^{12}$ bytes.

tasks during testing, introducing noise into the output. Syzkaller limits the kernel's coverage collection to only the current system call from the input payload, and coalesces the data after finishing executing the input payload.

2.2 Code Coverage in Fuzzing

Code coverage is the predominant execution feedback mechanism used by state-of-the-art fuzzers in most testing domains. It provides fuzzers with a control-flow-centric view of execution behavior introspection into the target program. Code-based coverage metrics can be mainly categorized into the following approaches: function coverage, line coverage, block coverage, and edge coverage. They track code-based program behavior and produce coverage on the following metrics, respectively: the number and identity of the functions executed; the number and identity of source-code-level lines executed; the number and identity of basic blocks visited during the program's execution, and the edges in the control flow graph visited during the program's execution.

Kernel fuzzers generally use block- and edge-based coverage metrics for novelty detection, as these provide more fine-grained feedback metrics regarding control-flow features during execution. Edge-based coverage presents a notable advantage, as it also reflects the direction of block transition, therefore distinguishing between block transition behaviors such as "A → B" and "B → A", where block coverage only knows that "A" and "B" have been visited, thus failing to discern between such behavior.

While code coverage is not fully representative of the states that the program goes through during execution, is considered to be somewhat efficient and effective for fuzzers to perceive some execution behavior changes, allowing inputs that trigger novel behavior to be preserved for further analysis and generation.

3 Motivation

Code coverage is the current established method of identifying execution behavior novelty in kernel fuzzers. However, its limitations are significant, in that many novel execution behaviors cannot be identified purely through the detection of a new code block or branch. There have been prior research towards addressing this issue, such as using state coverage in addition to code coverage. In this section, we discuss the limitations of pure code coverage, and the limitations of the current research works that attempt to address this problem.

3.1 Limitations in Pure Code Coverage

Code coverage has limits in execution state novelty detection. A change in the memory access pattern during execution can affect the overall outcome of a procedure, thus is a kernel execution state change, without altering its control flow, which is not perceptible using code coverage. In the domain of kernel fuzzing, we may have two invocations of the same system call, for instance *ioctl*, where the parameters used can alter kernel state differently, while not exhibiting a different code path. This affects the detection of novel kernel states and thus impedes high-quality input generation by increasing the difficulty of generating more effective input payloads for exploring kernel states and detecting bugs.

In theory, code and data exhibit *duality*, thus **data in a program's binary representation is also capable of reflecting its execution behavior**. In reality, data in a program's binary can contain features, including statically allocated data structures, runtime state flags, executable binary representations, opcode lookup tables, comparison operands, etc., whose coverage metrics can reflect their execution state. Code coverage is incapable of tracking the access behavior of such artifacts, as they are not represented as control-flow features during compilation or dynamic binary instrumentation, and in turn cannot provide execution feedback of such artifacts to the fuzzer during testing.

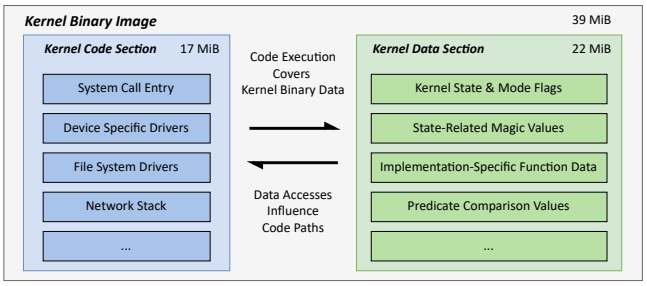

Figure 1: The composition of a Linux kernel's binary image, with relevant artifacts labelled. The kernel code under execution accesses the binary's data, which are mapped into the kernel's memory space, whereas the value of the accessed elements can in turn affect the control flow of the kernel's subsequent execution.

This is significantly evident in operating system kernels, as the sheer size of the compiled kernel and the data-relevant artifacts it contains present many opportunities for detecting new kernel execution behavior. To justify our claims, we present a quantitative analysis of the binary file composition of the latest Linux kernel upon writing, Linux v6.6.8, compiled to the *amd64* target with the default configurations for the OpenSUSE distribution. The results are shown in Figure 1, where we mainly outline the composition of sections containing either code or data. Note that this is not an exact figure, as immediate data are embedded within instructions, and compilers may insert data within the code section.

As shown in the figure, executable code consists of around 43% of the binary, while data uses around 57%. Adding data to coverage feedback potentially allows 2.3× coverage than pure code-based approaches to convey information for detecting novel behavior. Among the data, we have found various variables and data structures useful for further interpreting the kernel's execution state, allowing for a multitude of additional indicators, which cannot be effectively reflected by the use of incidents of control-flow artifacts, to be used for coverage feedback to the fuzzer for guidance for further input generation.

We demonstrate such limitations with the following example as shown in Figure 2. The blocks in the figure are excerpts from the source code of Linux kernel version 6.9. The code blocks that turn into code sections in the kernel binary are highlighted in blue, where those becoming static data are in green. In this case, consider the kernel fuzzer passing a request argument to the Linux

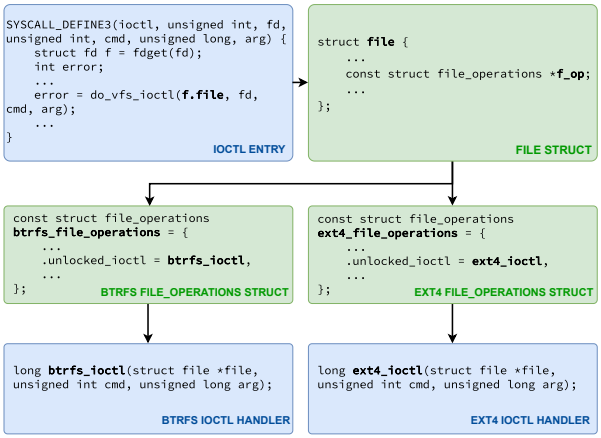

Figure 2: Example of the `ioctl()` system call's invocation chain, in which state information in the kernel's data affects the kernel's control flow. In this case, when `ioctl()` is called, as shown in the top-left, the pre-defined state transition, embedded as `file_operations` structures enclosed in each `file` structure for each `fd` argument passed in, determines the specific handler to run (`btrfs_ioctl()` for BTRFS or `ext4_ioctl()` for ext4). Pure code coverage makes it harder to execute different handlers, as the control code is covered but changes in the data accessed are indistinguishable.

system call `ioctl()`. As shown in the figure, the `f_op` member variable of the `file` structure, which is a `file_operations` structure, determines which code path it takes when a user program calls into `ioctl()`. Based on these in-data predefined state transitions encoded within the static `file_operations` instance for each file system and given the in-code transition logic defined in their processing logic, the fuzzer experiences difficulty in reaching different states in actual logic as the transition logic, i.e. the entry code in the top-left block, is easily covered, but the lack of feedback from the binary data components (`file_operations`) prevents the fuzzer from identifying differences in the referenced state transition table, thus is ineffective in generating effective request arguments, hindering its efforts in reaching different handlers, such as those in the lower two frames, thus reducing the states traversed within a fuzzing campaign.

3.2 Limitations in Kernel State Coverage

The limitations inherent in code coverage have been received with some research into *state coverage*, where works attempt to abstract the execution state of the program under test as feedback for fuzzers (e.g. IJON and StateFuzz). These approaches currently only attempt to model the states of specific variables, as an attempt to represent the running state of programs. We demonstrate an abstracted view of how *state coverage* and the actual *ground truth* calculate the set of all states ($\mathcal{S}$) as a function of current execution path in the program (P), expressed in and the individual state-discerning values (s) as a function of each variable in the program (v_k). We also express the set of all *independent* variables ($\mathcal{V}$) as a function of P.

$$\begin{aligned}(\text{Ground Truth}) : \mathcal{S}(P) &= s(v_1) \times s(v_2) \times ... \times s(v_m) \\ &= \Pi_{i \in \mathcal{V}(P)} s(v_i) \quad (1)\\ (\text{State Coverage}) : \mathcal{S}_c(P) &= s(v_1) + s(v_2) + ... + s(v_n) \\ &= \Sigma_{i=1}^{n} s(v_i) \quad (2)\end{aligned}$$

In theory, program states, within the context of the automata model, consist of the Cartesian product of the values of all independent variables in in the current path of the program and their execution context (Equation 1), whereas current approaches only discern between states of a variable (Equation 2). Such a technique would only increase the knowledge of certain variables, and significantly under-represent the program's actual execution behavior.

Additionally, these methods rely on manual labor to either identify or specify rules to identify all *state-indicating* variables, which scale poorly due to the complexity involved. Their state-discernible values of each variable also require specific domain knowledge or static analysis to determine, which is often error-prone, unsound, and incomplete rendering the results far from optimal.

Furthermore, as large codebases, such as operating system kernels contain many if not infinite automata states. Any attempt at precisely measuring the state of a kernel's execution will, very rapidly, encounter the problem of *state explosion*, rendering this method ineffective when fuzzing.

In the context of kernel fuzzing, these approaches have reversed the principles of what made code coverage applicable: **an effective set of artifacts that allow for efficient handling and effective execution behavior representation**, therefore do not present a sufficient feedback mechanism for effective kernel fuzzing. Reflecting on such pitfalls, we wish to avoid such intricacies in designing coverage feedback mechanisms that increase the scope of execution behavior conveyed to the fuzzer.

4 Design

We demonstrate the overall architecture of our proposed design in Figure 3. We use a generalized feedback-based kernel fuzzer architecture, which matches the runtime organization of state-of-the-art and openly-available tools (e.g. Syzkaller), to demonstrate our approach. The overall process starts from the "Fuzzing Executor" component in the Kernel Fuzzing triggering a fuzzing execution, and ends when the "Test Case Mutator" passes the next test case to the executor. Ideally, binary coverage intercepts memory accesses to static data and code loaded from the kernel's binary and compares their access patterns to detect novel behavior. In this process, we introduce design choices that are aimed at improving the approach's accuracy and efficiency.

(1) When instrumenting callbacks, we determine which memory uses have coverage semantics synonymous to visiting this basic block, and perform *Tracing Elision* to optimize out these tracing points (§ 4.1).
(2) We filter out variable loads not intended for sections corresponding to the kernel's binary (§ 4.2).
(3) We approximate addresses, assign context values, and estimate access lengths for valid accesses for both efficiency and accuracy, subsequently producing an access triple that holds the essence of the access (§ 4.3).

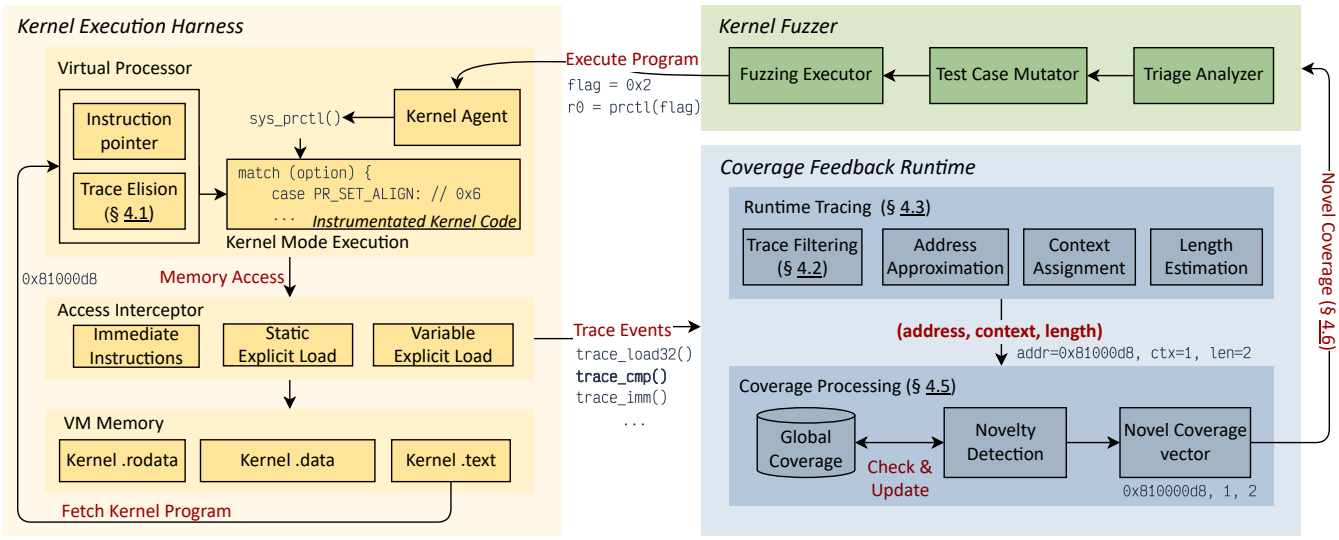

Figure 3: Overview of *Kernel Binary Coverage Feedback*'s usage during kernel fuzzing. The kernel is instrumented with tracing callbacks, which intercepts memory accesses and invokes tracing callbacks, and with redundant calls optimized out (§ 4.1). Trace Filtering is used to filter only accesses to regions from the kernel binary (§ 4.2). The *access triple* is produced through Address Approximation, Context Assignment, and Length Estimation (§ 4.3). The access is checked against historical coverage figures, where we detect novel coverage and save the triple if needed (§ 4.5). After finishing one round, the novel coverage is sent to the fuzzer along with code coverage for analysis and further generation (§ 4.6).

(4) The runtime takes the access triple and cross-references it with a global coverage store, which stores global historical coverage efficiently and accurately (§ 4.5).
(5) The novel access information is enqueued, and passed to the fuzzer upon finishing the execution, unifying code coverage in the process (§ 4.6).

The specifics of each design are described in the following sections. We begin with introducing the runtime monitoring and filtering operations, and then discuss the details regarding the runtime, especially how our approach takes a data access pattern and processes it for novelty detection, and then produces novel behavior data in the feedback to the fuzzer.

4.1 Access Interception and Trace Elision

The kernel binary is instrumented with tracing callbacks that invoke the coverage processing runtime's interfaces to capture the program's runtime data access patterns. A straightforward approach to record the data access information would be to find all instructions with memory accesses, and insert instrumentation calls that convey the syntactic information regarding this instruction to the runtime.

However, this loses many of the *spatial* and *temporal* execution behavior that would otherwise provide more fine-grained access information. We use a corollary of the advantages that block and edge coverage have over function and line coverage in delivering execution tracing information. Function and line coverage only represent coverage information one-on-one with static information present the `.text` section of the program's binary, and as such, loses the coverage of *control-flow features* offering *spatial* and *temporal* information, such as the direction and path of execution during a single run. This information is conveyed, partially, by block coverage, which presents spatial features, and more fully by edge coverage, which delivers more temporal information.

Using this insight gained from collecting control-flow features in code coverage, we interpret *data-flow features* from accesses to static data in the kernel's binary by instrumenting the *direct use-sites* of the static values.

We do not attempt to taint-analyze all locations where the static values are used, since, primarily, this does not provide substantially more useful information on the access of static data in the kernel, and additionally, this is very difficult to achieve with soundness and completeness guarantees. Instead, we use program transformation tools that covert the kernel's source code into *Single Static Assignment* (SSA) form, allowing us to identify where the static value is directly used. We limit our analysis scope to the confines of the function where the instruction belongs. While this may slightly reduce the completeness of information conveyed in the coverage metrics, our belief is that this is a good compromise between analysis complexity and accuracy. Our reasoning is that static values, when used as arguments or return values that go out-of-scope, it is rarely used in a context of a non-static value (e.g. static flags passed as function arguments are then compared with another static flag), thus losing little coverage accuracy. Furthermore, *Tracing Elision*, which we will discuss below, will render the rare instances irrelevant.

This is completed in a two-sweep analysis of the function's code, which is depicted in Figure 4.

As shown in the figure, during the first sweep, we collect all *def-sites* of static data, which are instructions that contain data access semantics, including instructions containing immediate values and

explicit loads, both static and variable, into the set S containing all such *def-sites*. We perform an over-approximation by inclusively recording all explicit memory loads with a variable base address, since their access targets cannot be determined statically and requires runtime filtering.

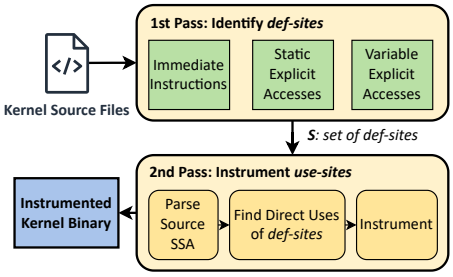

Figure 4: The Two-Sweep Analysis Workflow Diagram. The first pass takes the kernel's source files and looks for *def-sites* to add to the set S, which is passed to the second pass, where it finds *use-site* instructions containing direct usage of the *def-sites*. We instrument the relevant callbacks at the *use-sites* to track kernel binary access.

Then during the second sweep, we identify and instrument *use-sites* based on the SSAs in S, by finding instructions that use the SSA values as operands. We also assign a *use-site* ID based on the order of use in this function context, which is also conveyed to the runtime to discern between different uses of the same static data. For each such instance, we instrument an appropriate callback corresponding to the instruction's syntax. The second sweep is needed, as in some SSA languages, the ϕ *instruction* allows using an SSA that is declared after itself as its operand.

However, data accesses and uses are far more frequent than control flow altering instructions, therefore incurring a high overhead in theory. We draw on another observation that many tracing points are *semantically identical* to instrument the visit to the basic block through code coverage. An example would be a statically assigned integer being used in an arithmetic operation, whose access semantics are identical to covering its parent basic block. Eliminating such tracing points will be of no consequence to the accuracy of the coverage metrics provided. We devise *Tracing Elision* to remove such tracing points before executing the kernel. The tracing points that can be removed need to satisfy two conditions: first, their *def-sites* are static accesses; second, their *access length* is invariant in the *use-site*. The latter condition is important, as it is associated with *Length Estimation*, as discussed in § 4.3. Realistically, *Tracing Elision* is done in conjunction with instrumenting access points, where tracing callbacks are only then instrumented after the elision process determining the their necessities.

The instrumented tracing callback for each remaining access point conveys the syntactic information for both the *def-site* and the *use-site* instructions to the runtime. The former is for *Trace Filtering* and *Address Approximation*, while the *use-site* is for *Length Estimation* and *Context Assignment*, all of which will be discussed in the following sections.

4.2 Trace Filtering

During execution, the instrumented callbacks convey the access events to the runtime. However, as mentioned earlier, the instrumentation is an over-approximation of static data *use-sites*, as variable address explicit memory accesses can be directed to both static data and non-static sections.

We further filter access events triggered by uses of explicit load and store instructions during the kernel's execution to preserve only accesses to static data. The filter is derived from a description of the kernel's memory layout, provided during the kernel's initialization process. For instance, we can use the kernel's linker script to derive which locations sections containing static data are loaded into.

Using this information, we construct a bitmap filter, where each bit represents a unit of alignment for the underlying architecture, as loaders typically require the start of each section to be aligned. On *amd64* systems, the alignment is usually 8 bytes, thus each bit will effectively represent the access properties of a consecutive 64-bit region. On reading the memory layout description, the bits that correspond to contents in the kernel's static data sections (`.data`, `.rodata`, etc.) are assigned "1", where others are left as "0". The filter will validate each access event by converting the access address into the filter's index, by bitwise shifting the address $3 + N$ bits, where N is the logarithm to the alignment in bytes ($N = \log_2 8 = 3$ for *amd64* platforms).

The size of the filter, if naïvely implemented, can reach enormous sizes. For *amd64*, which uses 48-bit effective addresses, a direct allocation will produce a buffer 2^{42} bytes long. Fortunately, we observe that "1"s in the filter, i.e. valid regions of access, is globally sparse but locally dense, as sections are loaded into memory as a whole, and are contiguous in virtual addressing. Therefore, while we virtually map the same bit vector, we map memory pages that can contain the vector virtually, but assign their mapping to the same "0" page. Then, when assigning "1"s to the bitvector, we simply create new pages on demand.

4.3 Access Pattern Processing

We represent the *essence* of an access to the kernel's binary using a triple: *(address, context, length)*. The first element, *address*, indicates the approximate location in memory of the static data access's *def-site* to the kernel's binary, measured at the byte level. The second element, *context*, is a numerical estimation of the call stack state with the *use-site* address, allowing for more fine-grained access behavior interpretation. The last element, *length*, represents a best-fit calculation of the length of access at the *use-site*. The triple is produced by processing each tracing callback through *Access Approximation*, *Length Estimation*, and *Context Assignment*.

Access Approximation is the process of producing a unique *address* identifier for each access event. Intuitively, we can read the address of the traced access event's *def-site* directly to produce a straightforward address value.

However, not all memory accesses are explicit load/store instructions, where the data they access are also not directly conveyed through their instruction syntax. Immediate instructions are such an example, where it accesses data embedded within the instruction itself, requiring specific treatment to approximate addresses that represent the location of this access.

```
1 ; Subtraction with immediate
  sub eax, 0xDEADBEEF ; Hex: 2d ef be ad de, immediate offset by 1
3
  ; Bitwise shift with immediate offset
5 shl eax, 4 ; Hex: c1 e0 04, immediate offset by 2

7 ; XOR with immediate
  xor r11, 7 ; Hex: 49 83 f3 07, immediate offset by 3
```

Listing 1: Examples of immediate values in *amd64* assembly. Immediate values in the instruction and hexadecimal representation are highlighted.

The accurate locations of these immediate values are platform-, and, in some cases, instruction-specific. We demonstrate such instances of immediate values with an example in Listing 1, which is written in *amd64* assembly. In Line 2, the immediate value is offset from the start of the instruction by 1 byte, whereas for Line 5 and Line 8, they are offset by 2 and 3 bytes, respectively.

To provide an estimate of the immediate values' locations, we use the address of their embedding instructions as an approximation of their addresses. We achieve this by instrumenting a tracing callback, where the *def-site* addresses are assigned the program counter at each such instruction. When the execution reaches this instruction, the instrumented call invokes the library's tracing routines, which reads the instruction pointer from the virtual CPU, or a platform-specific, ABI-determined location. This is possible for achieving a precise approximation of the instruction's *address* value, as no other data can overlap with the location of this instruction.

Context Assignment mainly assigns a numerical value representative of the *use-site*, allowing us to distinguish between different uses of the same static value. While using the direct addresses of the *use-site* is possible, the complexity involved in designing storage structures that preserve accuracy and efficiency when taking two 64-bit addresses as keys is difficult. Instead, as we have already assigned *use-site* IDs for each instance during instrumentation, we directly use the value assigned as the context for each trace event.

Length Estimation takes the traced instruction and the approximated address and estimates the length of access. In many cases, the syntax of the instruction reflects the actual length of memory accesses. We use *estimations* of the actual length for cases where the actual memory access semantics of the instruction differ from its syntax.

We show two examples in Listing 2 of such cases in the Linux kernel. The first example is the `prctl` system call, which matches the argument `option` that is passed from user-space, to the numerous flags with a `switch` structure. Assuming that initially, an option is passed as a value of `0b11`, and is matched against `PR_SET_UNALIGN`, which is `0b101`. Assuming that equality comparisons begin with the least-significant bit and halt at the first differing bit, then in this comparison, only the least 2 significant bits are compared, instead of the 32-bit integer comparison that the syntax dictates. The second example is taken from `mount` system call, which masks the option bit vector with `MS_MGC_MSK`, which is `0xffff0000`, for use in a state comparison. The actual bits of the mask used in the bit-wise operation is dependent on the actual value of the other operand, and in this case, `flags`. Consider it to be `0x00010000`, then only one bit is used.

We estimate a semantically-accurate length of accesses for *use-sites* that are comparisons and bit-wise operations. Specifically, for

```
  // sys_prctl() definition
2 SYSCALL_DEFINE5(prctl, int, option, ...) {
      ...
4     switch (option) {
      ...
6     case PR_SET_UNALIGN: // Immediate value of 5
          error = SET_UNALIGN_CTL(me, arg2);
8         break;
      case PR_GET_UNALIGN: // Immediate value of 6
10        error = GET_UNALIGN_CTL(me, arg2);
          break;
12    ...
      }
14 }

16 // Invoked from sys_mount()
   int path_mount(const char *dev_name, struct path *path,
18        const char *type_page, unsigned long flags, ...) {
      if ((flags & MS_MGC_MSK) == MS_MGC_VAL)
20        ...
      ...
22    if (flags & MS_NOUSER)
          return -EINVAL;
24    if (flags & SB_MANDLOCK)
          warn_mandlock();
26    ...
   }
```

Listing 2: Examples of when the length syntax and semantics differ. Retrieved from the kernel's `prctl` and `mount` system call implementations.

equality and inequality operations, we use the number of matching bits in both operands; for partial-order comparisons, we count the number of matching bits starting from the most-significant bit to the first differing bit; for bit-wise AND operations, we also count the number of matching bits; for bit-wise XOR operations, we count the number of differing bits; and for other bit-wise operations, we use the syntactic length.

4.4 Access Semantics

We conclude on how memory accesses are traced, processed and abstracted, and categorize them into *access semantics* in Table 1. The first two columns, i.e. the *def-site* and *use-site* columns, are the properties of a specific memory access pattern, and define which procedure it uses for interception elision, access filtering, address approximation, context usage, and length estimation. All cases that our approach is interested in are listed in the table and are assigned with corresponding callbacks.

4.5 Coverage Storage and Novelty Detection

Global historical coverage is represented conceptually as a map, where it contains key-value pairs comprising values from the triple, with the *address* and *context* entries serving as keys to the map, and *length* as the keyed value. Each *address* and *context* indexes to the historically highest access *length*. As the addresses are expressed in bytes, we use one 8-bit integer to represent the coverage length.

To effectively manage coverage data for the entire kernel address space, we take advantage of the fact that code and data originating from the kernel's binary uses a very small and contiguous area in memory. A naïve approach is to use a linear vector of bytes, indexed by the address and context's hash with the length as the element. We improve on the naïve approach, by limiting the scope of the

Table 1: Interception analysis and processing procedures for different memory access patterns

def-site	*use-site*	Example	Elision	Filter	Address	Context	Length
Immediate	Comparison	`if (flag == 0b0101)...`	No	No	*def-site* Program Counter	*use-site* ID	Comparison Estimation
	Bit-wise Operation	`if (flag & 0b0101)...`	No	No	*def-site* Program Counter	*use-site* ID	Bit-wise Estimation
	Other Usage	`return x << 4;`	Yes	-	-	-	-
Explicit Load (Static)			Same as Immediate				
Explicit Load (Variable)	Comparison	`if (num == array[i])...`	No	Yes	*def-site* Access Address	*use-site* ID	Comparison Estimation
	Bit-wise Operation	`if (flag | bitvec[i])...`	No	Yes	*def-site* Access Address	*use-site* ID	Bit-wise Estimation
	Other Usage	`int num = buf1[i] * buf2[j];`	No	Yes	*def-site* Access Address	*use-site* ID	Syntactic Length of Access

buffer to the range of possible code and data addresses. The range is readily available by reading the section headers in the kernel binary.

For Linux, we use *readelf* to find the top- and bottom-most addresses of all code and data sections. The addresses can be expressed using the least-significant bits of their addresses (26-bits on *amd64*), thus we use this as the index.

We further provide better accuracy than the naïve approach by setting the indexed variable to a pointer, which references a *def-site* specific buffer that lengths corresponding to contexts. The *length* should be in a range of $[0, 128]$. This approach is beneficial: first, memory access lengths are frequently multiple bytes, then this allows for a single element to contain access metrics for the following 16 bytes; and second, we can discern between overlapping accesses to the same region. We also use the most-significant bit as an indication of full access, therefore bypassing many checks and arithmetic operations when processing tracing events.

As small and frequent allocations are inefficient, we maintain multiple pools containing fixed-size buffers, where the sizes are multiples of the L2 cache line size, which is generally 8 bytes. The pools are pre-allocated and maintained using a free-list, while a pointer for each pool points to the next available buffer. Additionally, the *address*-indexed buffer is initialized to "0" and is initially mapped to one physical page. Specific buffers for each *address* entry are allocated from the pool and increased with size on-demand.

When tracing events occur and the previous steps produce a coverage triple, we check the historical coverage data and determine if any novel behavior occurs. Within one execution cycle, all novel memory accesses will be added into a *novel set*, where each entry contains the triple of the novel memory access. We then update the global coverage accordingly for future executions.

In contrast to other collection approaches, where current coverage is stored into a local bitmap, and scans for novel behavior commence post-execution and concern the entire buffer, our method provides better efficiency. Theoretically, traditional methods use a time complexity of $O(n) + O(k)$, where n is the bitmap size and k is the number of tracing events. Our approach's complexity, in contrast, is $O(k)$, as there is no need for post-execution scans.

4.6 Coverage Feedback to Kernel Fuzzers

After one round of execution, we collect novel behavior coverage from both the *novel set* and code coverage. We merge the sets and treat code coverage as a special type of memory access. Using the program counters of blocks visited, we use a sliding-window method with a width of 2, through which define the preceding block as the *def-site*, and the current block as the *use-site*. The lengths are set as one single bit to avoid interfering with actual memory access traces. The triples are then inserted into the *novel set*, which is then conveyed to the fuzzer as execution feedback. This allows the fuzzer to use the same processing techniques to triage and analyze, reducing such overheads and design complexities.

5 Implementation

We implement our designs in a kernel binary coverage feedback tool named KBinCov. To facilitate our evaluation process, we also implement multiple variants that assist us in assessing the effectiveness of our design choices. In addition, we also follow the reference designs IJON to implement versions suitable for providing whole kernel coverage feedback, kIJON, for comparison.

5.1 KBinCov Implementation

We implement KBinCov upon the state-of-the-art kernel fuzzer Syzkaller with QEMU as the virtualization platform. Syzkaller and QEMU mainly support using *Kernel Virtual Machine* (KVM) for same-platform guests to achieve near-native execution speeds, and its *Tiny Code Generator* (TCG) infrastructure for foreign architectures. To support both KVM and TCG execution modes, we provide two reference implementations: KBinCov-KVM and KBinCov-TCG, which correspond to collecting binary coverage on QEMU running on KVM mode and TCG mode, respectively.

KBinCov-KVM uses LLVM's *SanitizerCoverage* (SanCov) framework with extensions to support additional instrumentation points. The static analysis for determining all necessary trace points is implemented as an LLVM Pass that runs ahead of SanCov. The runtime itself runs within the kernel, and interacts with Syzkaller through a *debugfs* interface similar to `kcov`.

KBinCov-TCG modifies TCG's dynamic translation process and hooks the point after lifting to TCG's Intermediate Representation finishes to perform static analysis and instrumentation. In this case, the runtime runs in the fuzzing host, and interacts with the fuzzer through direct shared memory.

5.2 Variants for Comparisons

We also implement the following variants of KBinCov for comparative evaluation purposes. We show an outline of our implemented variants below in Table 2. Each variant has either 1) one design

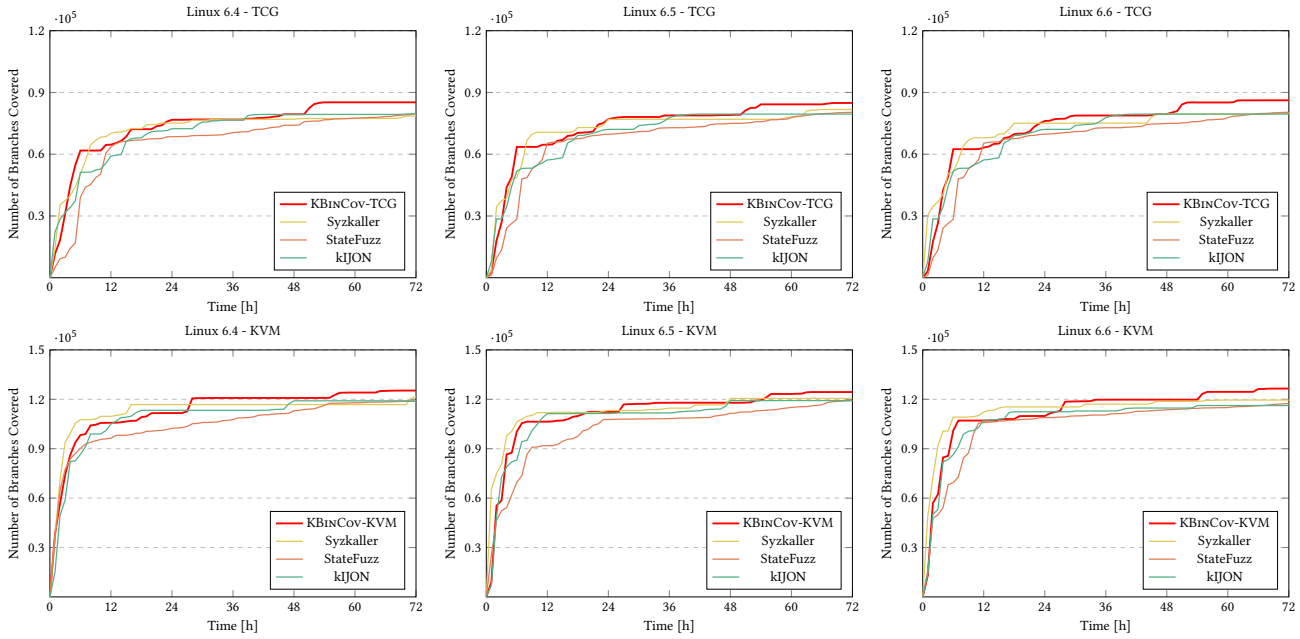

Figure 5: Branch coverage statistics of Syzkaller with KBinCov, Syzkaller (kcov), StateFuzz, and kIJON for Linux kernel versions of 6.4 to 6.6 for 72 hours. Statistics for both KVM and TCG modes are shown. Higher statistics are better.

choice reversed (KBinCov-NoElision to KBinCov-NaïveArray), or 2) only one type of feedback collection method enabled (KBinCov-Imm to KBinCov-Variable). As these are intended only for evaluating the effectiveness of our approach, their implementations are based on KBinCov-KVM only.

Table 2: List of KBinCov evaluation variants and their corresponding design changes.

Variant	Design Difference
KBinCov-NoElision	No Tracing Elision During Instrumentation
KBinCov-NoTraceFilter	Disable Filtering of Non-Static Accesses
KBinCov-PrecImm	No Address Approximation During Processing
KBinCov-NoLenEst	No Access Length Estimation During Processing
KBinCov-NoContext	No Context Calculation During Processing
KBinCov-NaïveArray	Naïve Coverage Storage Implementation
KBinCov-Imm	Only Immediate Value Access Collected
KBinCov-Static	Only Static Memory Access Collected
KBinCov-Variable	Only Variable Memory Access Collected

As the original version of IJON is designed for coverage feedback on userspace programs only, we manually implemented a version of IJON, named kIJON, for comparative evaluation purposes with our approach. Specifically, we re-used its original variable annotation method, but replaced their runtime implementations with custom-written routines, which are intended for sending the coverage feedback to the kernel fuzzer in conjunction with the code coverage metrics for execution novelty detection.

6 Evaluation

We wish to understand the effectiveness of our approach in the sense of assisting kernel fuzzers achieve better performance. Additionally, we are especially interested in the qualities of our design choices. To this end, we list the following four evaluation criteria to assess: overall effectiveness in coverage improvements (§ 6.2), overall efficiency and accuracy (§ 6.3), effectiveness of our individual and component-wise design choices (§ 6.4), and real-world bug detection capabilities (§ 6.5).

6.1 Experiment Setup

Our experiments are conducted on a server running dual-socket AMD EPYC 7742 CPUs, 256GiB of DDR4 RAM, with Debian Bookworm as the host operating system. All fuzzing instances are containerized with reasonable resource constraints.

The kernels used in our evaluation are Linux v6.4, v6.5 and v6.6.8, all recent releases of the kernel. The tarballs of the kernel are downloaded from kernel.org and its affiliated mirrors, with checksums matching the officially provided values. The kernel compilation configuration used is based on Debian's kernel configuration file, where specific parameters (e.g. `CONFIG_KCOV=y`) were adjusted for vanilla Syzkaller to achieve parity with Syzbot's configuration [26]. The target architecture for compilation is *amd64*.

We evaluated the performance of KBinCov integrated with Syzkaller, and added vanilla Syzkaller, StateFuzz, and kIJON with Syzkaller for comparison. The version of Syzkaller used is Git commit *83b32fe*, the latest commit prior to evaluation. The version of StateFuzz is Git commit *e4fd485*, the current open-source version, with our own bugfix patches applied. The kernels for testing are hosted under both KVM and TCG for evaluation of the two

implementations. Each fuzzing instance is allocated 2 concurrent fuzzers, each with 2 executors and 4GiB of VM memory. We enabled KASAN [10] and KCSAN [11] for alternating instances, ensuring that each fuzzing setting has both types of instances running.

The coverage experiments are conducted for 72 hours. All bugs listed were found in this experiment. Each experiment was repeated 5 times to reduce statistical errors. We used statistical testing (Mann-Whitney U Test) to establish the significance of the evaluation data.

6.2 Overall Effectiveness

To understand the effect of using binary coverage in kernel fuzzing, we deployed six evaluation scenarios, specifically the cartesian product of the three Linux versions, v6.4, v6.5 and v6.6, through using both KVM virtualization and TCG translation. In each scenario, we compared the coverage metrics of Syzkaller using the following coverage feedback techniques: KBinCov, Syzkaller's code coverage through kcov, StateFuzz, and kIJON. During this campaign, we collected memory usage and execution throughput metrics by polling every 5 minutes. After the fuzzing run, we collected the seeds collected by each instance to analyze KBinCov and the comparison tools' ability to discern between execution states marked as novel by the seeds.

6.2.1 Code Coverage. The 72-hour experiment results on code coverage metrics over time for KBinCov, Syzkaller, StateFuzz and kIJON are shown in Figure 5. Overall, our implementations of KBinCov-TCG and KBinCov-KVM achieve better coverage statistics than Syzkaller's kcov, StateFuzz, and kIJON over 72 hours on both TCG and KVM execution modes. This is no small feat, as the TCG and KVM implementations differ greatly due to the different toolchains used to construct our method. Being able to challenge not only traditional code coverage, but also state coverage tools further demonstrates the versatility and robustness of our design.

The statistics are listed as follows: on QEMU-KVM, KBinCov-KVM achieves an average coverage of 124141 branches on the three versions, with a standard deviation of 215. On QEMU-TCG, KBinCov-TCG achieves a branch coverage metric of 83532, with a standard deviation of 443. This puts KBinCov on a 7%, 7%, and 9% lead compared to Syzkaller (KVM average=116020 w/ stddev=77, TCG average=78067 w/ stddev=253), StateFuzz (KVM average=116490 w/ stddev=541, TCG average=77952 w/ stddev=253), and kIJON (KVM average=113891 w/ stddev=79, TCG average=76635 w/ stddev=121).

An obvious trend for KBinCov in both execution modes is that KBinCov's coverage metrics tend to overtake the comparison tools relatively late into the campaign. This can be mainly explained through an increase in overhead for KBinCov, as we will delve into shortly, thus fewer fuzzing cycles executed as a whole, and an increase in novel states found, increasing the need for tasks such as triage, seed maintenance, etc. While this may have initially put KBinCov at a disadvantage, the new execution states that our method can detect contributes to KBinCov's eventual lead.

Under QEMU-KVM, we find that the coverage statistics are significantly higher for the tools than on QEMU-TCG. Our investigations into the configurations show that the emulator enabled a different set of devices under the two modes, thus affecting the number of modules the kernel loads during initialization, and in turn resulting in a difference in the figures when saturated. Therefore, it is not indicative of the limitations of KBinCov-TCG.

6.2.2 Binary Coverage. We also measured KBinCov's effectiveness in actually covering the kernel's binary by comparing the binary coverage statistics with that of Syzkaller, StateFuzz, kIJON. Like the previous metric, we collect the their seeds at the end of their respective 72 hour trials and use a dry-run of Syzkaller with KBinCov to identify unique seeds, and then calculate their accumulated binary coverage statistics by coalescing the coverage information of each individual seed, where overlapping accesses are counted only once, similar to that of code coverage. The unit of the statistics are measured in KiB of memory read from the kernel's binary loaded into memory during execution over their respective 72-hour trials.

Table 3: Binary Coverage Statistics of KBinCov on the tested Linux kernels, compared to that of Syzkaller, StateFuzz, and kIJON. Units are in KiB of binary accessed

Coverage	Linux-6.4		Linux-6.5		Linux-6.6	
Mechanism	KVM	TCG	KVM	TCG	KVM	TCG
KBinCov	15730	9856	14262	9325	17273	12152
Syzkaller (kcov)	7536	6465	8102	6327	7327	5982
StateFuzz	11780	7589	10510	6621	13672	8808
kIJON	8791	7021	8342	7142	8804	8319

The results are shown in Table 3. As we observe from the statistics, KBinCov achieves a significantly higher binary coverage statistic than the comparison. Specifically, KBinCov shows a 76% to 108% increase on QEMU-KVM and 47% to 103% increase on QEMU-TCG over Syzkaller's vanilla kcov implementation, a 26% to 35% increase on QEMU-KVM and a %30 to 40% increase on QEMU-TCG over StateFuzz, and a 71% to 96% increase on QEMU-KVM and a 30% to 46% increase on QEMU-TCG over kIJON.

These statistics demonstrate that: 1) our approach is effective in providing binary coverage feedback, where the semantics of such feedback differ greatly from pure code coverage, as well as state coverage from previous research; and 2) our design and implementation is effective in assisting kernel fuzzers to achieve a significantly greater statistic than the comparison with the same time constraint.

6.2.3 Binary Coverage vs. Code Coverage. We further compare the statistics of binary coverage, i.e. Syzkaller with KBinCov, with code coverage, i.e. vanilla Syzkaller, to demonstrate the similarities and differences between the two execution feedback mechanisms. The results are shown in Figure 6.

As shown in the plot, the growth of binary coverage is different compared to that of code coverage, as there are durations during which the statistics for binary coverage grows while code coverage remains constant. Additionally, we also see that when code coverage grows, especially when code coverage *jumps*, such as when an entire module is executed for the first time, we see that KBinCov's statistics also show a corresponding significant increase, indicating that code coverage metrics are reflected as binary coverage. These characteristics show that on the one hand, binary coverage captures all code coverage metrics, which is in accordance with our design,

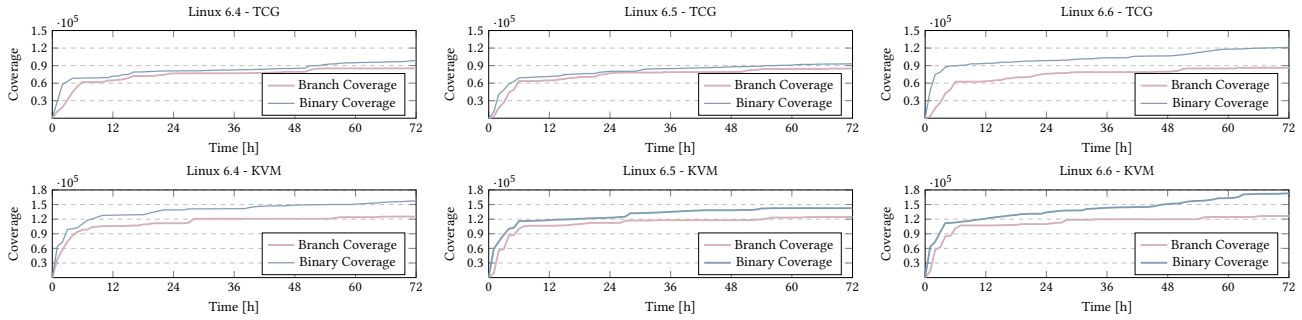

Figure 6: Binary coverage statistics (10^2 Bytes Data Accessed) compared to branch coverage statistics (Branches) of Syzkaller with KBinCov for Linux kernel versions of 6.4 to 6.6 for 72 hours. Statistics for both KVM and TCG modes are shown. The two coverage statistics exhibit different growths and binary coverage grows even when branch coverage does not.

and on the other hand, discerns more execution state changes than code coverage, which is our eventual design goal.

6.2.4 Summary. Considering both that the code coverage metric does not account for the new novel behavior that our method delivers, while KBinCov achieves better *code coverage* statistics, and that KBinCov achieves significantly higher statistics when using *binary coverage* as the evaluation statistic, it is certain that KBinCov is capable of achieving better coverage feedback to the fuzzer, leading the comparison tools in coverage statistics consistently.

6.3 Accuracy and Efficiency

All major design considerations for kernel binary coverage were about the compromise between accuracy and efficiency. Here, we wish to understand how well we reached our goals as a whole, by breaking down the overhead statistics, both in memory usage and runtime slowdowns, in addition to analyzing whether the compromises satisfy fuzzing needs.

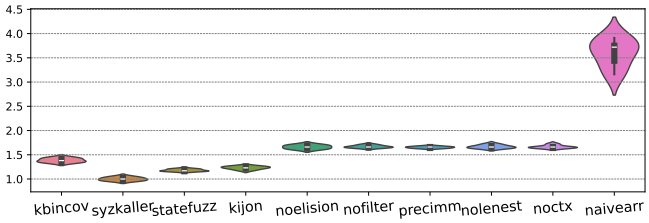

Figure 7: Memory Overhead Analysis of KBinCov-KVM, Syzkaller (kcov), StateFuzz, and kIJON, along with KBinCov's variants. We use Syzkaller as the baseline. Lower is better.

6.3.1 Memory Overhead. We first analyze memory overheads potentially from implementing kernel binary coverage and compare the increase in memory usage against Syzkaller with kcov, in conjunction with StateFuzz and kIJON. Note that we measure the entire fuzzing harness's memory footprint, as coverage feedback cannot be easily measured alone. We set Syzkaller as the baseline and compare all tools with modified coverage feedback with it to understand the overhead increases that each approach delivers. In addition, we add KBinCov's variants in this Using the memory usage metrics sampled over the 72-hour fuzzing campaign, we average the values for each scenario, and plot the 5 average values in Figure 7. As is apparent in the plot, KBinCov's memory usage increase is noticeable, but acceptable, as it is only around 35% more memory usage, 10% higher than StateFuzz's, and 8% higher than kIJON.

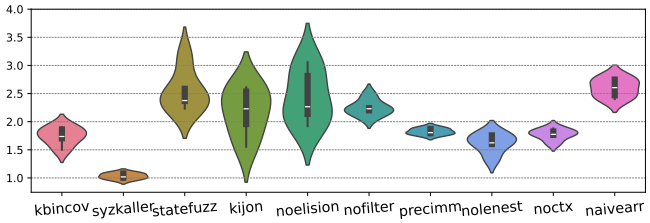

Figure 8: Runtime Overhead Analysis of KBinCov-KVM, Syzkaller (kcov), StateFuzz, and kIJON, along with KBinCov's variants. We use Syzkaller as the baseline. Lower is better.

6.3.2 Runtime Overhead. We then analyze the runtime overhead increases of using kernel binary coverage. To assess the added cost from incorporating *full kernel binary coverage* into kernel fuzzing, and through using the execution data collected during fuzzing, we measure the metric of total number of fuzzing executions in 72 hours to assess the actual performance of the fuzzer, and in turn compare the runtime overhead between the tools to understand the efficiency of kernel binary coverage. Similarly, we set Syzkaller's average performance as a baseline and observe how the different approaches affect runtime overheads. The results are plotted in Figure 8. It is evident that, while KBinCov exhibits obvious overheads of up to 90%, its runtime overhead statistics are, in comparison, on-par or better than StateFuzz or kIJON, which may reach over 2× execution time compared to vanilla Syzkaller. In addition, as previous statistics have demonstrated, fuzzing effectiveness cannot be measured entirely on the execution throughput figure, but rather a comprehensive set of factors. Syzkaller's faster execution speed, and, in contrast, StateFuzz's and kIJON's slower approach, both yielded inferior figures in coverage statistics, thus demonstrating that our approach reaches a better balance than the comparison in fuzzing efficiency.

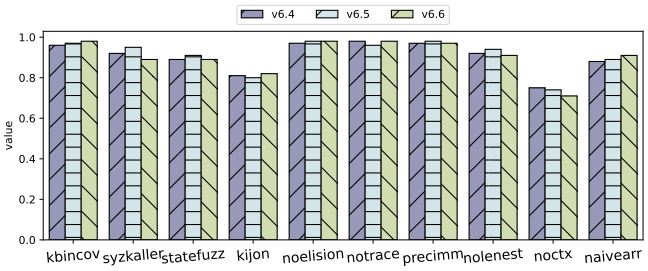

Figure 9: Recall rates comparison for KBinCov-KVM, Syzkaller (kcov), StateFuzz, kIJON, and the KBinCov variants. Higher is better.

6.3.3 Accuracy. To understand the accuracy of our approach in comparison to the other tools, we use the *recall rate* metric, which, in this case, refers to the proportion of seeds that the fuzzer can find novel behavior in, and as such identifies more or less behavioral characteristics in the kernel. We obtain this metric for the comparison tools using the following procedure. First, we collect the seeds accumulated by each fuzzing instance during the fuzzing campaign. We then perform a dry-run on each seed set for each tool and count the number of seeds that the fuzzer recognizes as "interesting" based on detecting novel behavior through using coverage feedback. For each seed set, we calculate a set-local recall rate that reflects how large of a proportion a specific coverage feedback tool can be identified as novel. Finally, we calculate a weighted average for each coverage tool, with weights determined by the size of the individual seed set.

We show the results in Figure 9. As is evident in the graph, KBinCov's recall rate is not only significantly higher than both vanilla Syzkaller, increasing by 7 percentage points, but is also higher than that of StateFuzz and kIJON by a significant margin, measuring at around 10 and 20 percentage points. While it is possible that, as KBinCov covered the most branches, it has a lead in the number of seeds contributed to this cause, it is more probable that these approaches, while focusing on the specific values of variables, did not put enough emphasis on using static program variables in the kernel's binary, thereby bypassing many state information. Nevertheless, we find that KBinCov's accuracy exhibits excellent performance, as demonstrated by its high recall rate.

6.3.4 Ablation Tests for Collection Criteria. We further conduct an ablation test to better understand the effect on coverage statistics and fuzzing performance as a whole that each collected metric contributes, as outlined in Table 1. Specifically, we are concerned with how collecting only either immediate values, static explicit loads, and variable explicit loads affect fuzzing performance, i.e. code coverage statistics. We use KBinCov, KBinCov-Imm, KBinCov-Static, KBinCov-Variable to denote the variants of KBinCov that are, respectively, KBinCov's original version, KBinCov with only immediate value collection enabled, KBinCov with only static value collection enabled, and KBinCov with only dynamic value collection enabled. The code coverage (branch) of each instance is calculated post experimentation and the average of its performance on the Linux kernels v6.4, v6.5 and v6.6 are taken. The results are shown in Figure 10, where the statistics for individual kernels are shown as bars, while the average statistics are shown as the line.

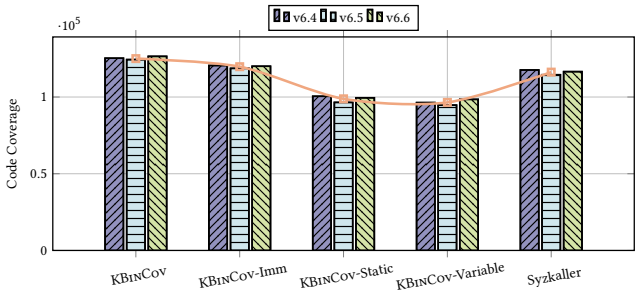

Figure 10: Results of ablation test regarding each individual coverage metric compared with the original version of KBinCov and Syzkaller. Higher is better.

As shown in the figure, KBinCov leads Syzkaller in coverage metrics by 7%, which is consistent with our previous findings regarding code coverage gains. For the individual collection metrics, we find that collecting only immediate values, i.e. KBinCov-Imm, delivers lower coverage statistics than KBinCov, where the original version leads this case by 5% but has a higher statistic than vanilla Syzkaller. This is because all code coverage statistics are mapped into immediate value accesses, thus providing feedback function parity with code coverage in this regard, whereas other immediate values, such as those used in branching predicates, allows for more fine-grained execution state feedback, which allows the fuzzer to save more meaningful seeds, and gain more coverage over time. Without such mapping, as in the case of KBinCov-Static and KBinCov-Variable, we find lower performance when compared to either KBinCov or Syzkaller (KBinCov performs 26% and 30% better than the two cases, respectively). While static explicit loads and variable explicit loads are commonplace, the data shows that they cannot collect feedback even on the degree of code coverage themselves.

6.3.5 Summary. We conclude that KBinCov achieves an excellent balance between accuracy and efficiency as a whole, performing better than the baseline approach and the reference state coverage approaches, further demonstrating the effectiveness of our approach.

6.4 Design Choice Effectiveness

We further analyze the effectiveness of our design choices on a case-by-case basis according to the evaluation statistics shown in Figures 7, 8, and 9.

6.4.1 Tracing Elision: KBinCov-NoElision does not compromise on accuracy, as the recall rates are very close to the reference implementations of KBinCov. This is to be expected, as KBinCov-NoElision generates a super-set of KBinCov's trace points. For efficiency, removing tracing elision does not significantly impact memory footprint, even though the code sections in the kernel have grown bigger as a result of the excess instrumentation. However, its performance exhibits severe degradation, seeing an increase at around 30%.

6.4.2 Trace Filtering: We observe that disabling trace filtering does not have a significant impact on the accuracy of coverage feedback, as demonstrated by the close recall rates that the reference design and KBinCov-NoTraceFilter both exhibit. This is also to be expected, as disabling tracing does not affect the effective tracing callbacks that detect the novel behavior. While its memory footprint does not increase significantly, runtime efficiency is moderately affected, as removing tracing filters but increasing the number of callbacks executed results in a net increase in execution time on tracing callbacks.

6.4.3 Address Approximation: Removing address approximation requires an additional step for immediate instructions, where the precise location of an immediate value access is calculated. While, in theory, this process will provide a address with byte-level precision, we do not see significant improvements in the recall rate, demonstrating that such approximation provides sufficient precision to identify different execution states. In contrast, the runtime overhead shows a slight increase, whereas memory usage shows no significant improvement.

6.4.4 Context Assignment: The removal of context assignment shows obvious drops in recall rates, indicating that the effectiveness of our approach is hindered without context information, whereas efficiency has not improved, either in memory usage or runtime overheads, thus its inclusion is a purely positive decision. This is to be expected, as the removal of *use-site* context will not allow the fuzzer to discern between different uses of the same kernel data.

6.4.5 Length Estimation: Removing length estimation also degrades fuzzing effectiveness, which is reflected by its slightly lower recall rates. Efficiency gains are also not apparent, with only a slight reduction in average runtime overhead. This is also to be expected, as many behavioral characteristics, such as comparisons and bit-wise operations, will be lost and not counted as novel behavior.

6.4.6 Coverage Storage: Using the naïve approach towards storing coverage data results in disastrously bad performance statistics, specifically with a significant increase in both memory and runtime overhead. Therefore, it is meaningless to analyze its accuracy as this choice is simply not practical for actual kernel fuzzing use.

6.4.7 Summary: Using the assessment above, we demonstrate, through the statistics of altering one component only, that all design choices in our approach all contribute towards improving either or both the accuracy and efficiency of reflecting the coverage statistics of the kernel under test.

6.5 New Bugs Detected

During testing, KBinCov assisted Syzkaller in finding 21 previously undetected bugs in the Linux kernel versions v6.4, v6.5, and v6.6, which are listed in Table 4. In contrast, of the total 21 found with Syzkaller and KBinCov, Syzkaller using (kcov) found 4, StateFuzz found 4, and kIJON found 2. All listed bugs have been responsibly reported with anonymous accounts. Interestingly, Syzkaller under other coverage feedback methods, i.e. code coverage, StateFuzz and kIJON, either could only find bugs previously detected by Syzbot [26], or detected the bug with time consistently exceeding that of KBinCov. We believe that this is due to most of the bugs persisting in the kernel requiring many conditions to meet before being triggered, where our approach allows for more fine-grained feedback of the execution of such conditions, thus being able to discover the buggy condition sooner.

Table 4: List of the 21 new bugs found by Syzkaller using KBinCov as feedback during the evaluation.

ID	Location	Description
1	drivers/hid/usbhid/hid-core.c	UBSAN: array-index-out-of-bounds in usbhid_parse
2	net/netlink/af_netlink.c	BUG: soft lockup in ipv6_list_rcv
3	kernel/rcu/tree.c	BUG: soft lockup in rcu_core
4	kernel/events/core.c	possible deadlock in __perf_install_in_context
5	fs/ext4/move_extent.c	WARNING: locking bug in __ext4_ioctl
6	net/9p/protocol.c	memory leak in p9pdu_readf
7	kernel/sched/core.c	INFO: task hung in perf_event_free_task
8	kernel/entry/common.c	INFO: rcu detected stall in syscall_exit_to_user_mode
9	fs/open.c	BUG: soft lockup in sys_openat
10	mm/madvise.c	BUG: soft lockup in sys_madvise
11	arch/x86/kernel/time.c	KASAN: stack-out-of-bounds Read in profile_pc
12	fs/buffer.c	KASAN: out-of-bounds Write in end_buffer_read_sync
13	fs/ntfs3/inode.c	KASAN: global-out-of-bounds Read in ntfs_iget5
14	fs/btrfs/free-space-tree	kernel BUG in populate_free_space_tree
15	net/ipv4/tcp_input.c	BUG: soft lockup in ip_list_rcv
16	fs/ext4/mballoc.c	divide error in mb_update_avg_fragment_size
17	fs/ext4/mballoc.c	KCSAN: data-race in ext4_mb_regular_allocator / mb_mark_used
18	fs/kernfs/inode.c	KCSAN: data-race in kernfs_refresh_inode / kernfs_refresh_inode
19	mm/page-writeback.c	WARNING in ext4_dirty_folio
20	drivers/tty/tty_ioctl.c	KCSAN: data-race in n_tty_lookahead_flow_ctrl / tty_set_termios
21	fs/ntfs3/record.c	WARNING in mi_init

Qualitative Analysis: To understand how using KBinCov allowed kernel fuzzers to trigger the the bugs found, we reproduced the bugs under the conditions where they were discovered, and verified the following criteria: 1) whether Syzkaller's coverage feedback mechanisms, i.e. *kcov*, were able to reach the bug's location, and, within 7 days of testing, can eventually trigger the bugs. Our analysis that the bugs can be classified into the following three categories, according to whether code coverage is capable of reaching the relevant bug's location in the kernel, and its ability to trigger the bug eventually: 1) KBinCov uniquely covers the bug's location in code (bugs 4, 6, 9, 12, 13, 16, and 21), and thus code coverage is incapable of assisting the fuzzer in triggering the bug; 2) the buggy code is covered by code coverage, but the fuzzer cannot trigger the bug in the allotted amount of time (bugs 1, 2, 7, 8, 10, 11, 14, 17, 18, and 20), therefore demonstrating that KBinCov uniquely provides the fuzzer with required feedback to progress towards the buggy state; 3) KBinCov helps fuzzers trigger the bugs within less amounts of time than using code coverage (remaining new bugs), thus the new execution states identified by KBinCov allows fuzzers to explore the kernel's state space more efficiently, thus identifying buggy states faster. For the first case, KBinCov uniquely covers the buggy code, while for the latter two, code coverage reaches the location, but is handicapped in triggering the bugs,

Summary: Through finding new and real-world bugs in the Linux kernel, we show that our approach is readily applicable and can assist kernel fuzzers in detecting bugs.

7 Discussion

7.1 Threats to Validity

Our design and evaluation process, however careful, may include inaccuracies, in the following possible cases.

First, our implementations of previous research IJON may differ from its specific reference implementations, if applicable, and may be a source of inaccuracies, as its actual implementations are not directly applicable to kernel fuzzing at the time of writing. We have implemented kIJON to the best of our abilities, and have matched the performance of our implementations with its original papers.

Additionally, our evaluations show that KBinCov performs favorably when compared with state coverage approaches StateFuzz and IJON. However, as our approach is not a direct attempt to address the issue of tracking state transitions, there are cases where this method will have difficulty being applied, such as interpreters and JIT compilers that read foreign programs. In theory, state-based coverage in theory can achieve better coverage metrics, but their current implementations are also ill-equipped to do so.

7.2 Generality of Method

The essence of kernel binary coverage is not limited to assisting in kernel fuzzing. Our approach, can be tailored to be used in userspace application testing through a number of steps, including migration of kernel-space instrumentation to userspace, implementing a userspace-specific coverage collection and processing runtime, etc. This is because while operating systems kernels and userspace applications differ greatly in form and function, both kernels and userspace programs contain code and data sections that have highly similar implications towards improving fuzzing feedback.

Furthermore, all program testing methods can benefit from utilizing binary coverage, regardless of the target program's size or complexity, as our method does not remove the semantics of code coverage, but builds upon such feedback through mapping such behavior as memory accesses, while increasing the execution information collected through collecting the other memory accesses, such as static explicit accesses.

8 Related Work

8.1 Kernel Fuzzing

Syzkaller is a state-of-the-art kernel fuzzer designed for general purpose operating systems such as Linux, macOS, and Windows. Aside from Syzkaller, there has been much research effort [23, 29, 33] on developing efficient kernel fuzzing technologies.

First, efficient kernel fuzzing requires highly effective input generation methods. HEALER [25] is a Syzkaller-inspired kernel fuzzer that uses a relation-learning technique to continuously learn the relations (i.e. correlation) between system calls at runtime and then generate high-quality system call sequences. Moonshine [17] uses seed distillation on real-world traces to produce quality seeds. KSG [24] uses static analysis to find the entry points of specific kernel modules and generate specific system calls grammar. There are also attempts at avoiding the use of such grammar, a.k.a. no-grammar fuzzing. FuzzNG [3], for example, tackles this problem by inferring the interface of system calls using real-world examples and runtime information.

Apart from common memory bugs in system call interfaces, there are efforts in finding other types of bugs and bugs located in different kernel modules. Razzer [12] targets kernel race bugs by directing fuzzing probable race locations and interleave kernel threads to trigger race bugs. Certain components in the kernel, like driver modules, tend to receive less scrutiny. Saturn [30] targets to cover the entire handling chain throughout the USB communication. Furthermore, there are works that analyze the detected bugs' severity. SyzScope [34] aims to analyze thousands of seemingly "low risk" kernel bugs and found hundreds that were, in fact, of high severity. GREBE [13] proposes an object-driven kernel fuzzing technique to explore various contexts to trigger reported bugs and detect various error behaviors.

Efficiency is just as important as input generation techniques when your test target is hosted in a virtualized environment. kAFL [20] is a kernel fuzzer that utilizes hardware extensions in modern processors to greatly increase the efficiency in coverage collection processes, therefore effectively increasing its execution throughput. Horus [14] leverages accelerates fuzzing-related data transfers, thereby improving overall fuzzing efficiency.

Different OS targets are also an area of research interest. For example, embedded operating systems have also received attention from various researchers in recent years. SFuzz [5], RtKaller [21], Gustave [8], and Tardis [22] are all works which tackles the problem of efficient and effective testing of embedded operating systems.

Our work is designed to assist in enhancing the feedback mechanisms of current kernel fuzzers. Therefore, the related works in kernel fuzzing listed above all can potentially benefit from adapting KBinCov to their execution feedback workflows.

8.2 Execution Feedback Mechanisms

Current state-of-the-art fuzzers are designed to utilize either single or multiple execution feedback mechanisms [15, 16, 18, 28] to extract and interpret the underlying details of a program's execution behavior, which is then used to guide input generation, bug reproduction, execution calibration, branch predicate solving, etc.

Taint analysis is frequently used by fuzzers that employ hybrid fuzzing techniques to gain additional insight into how the generated inputs affect its execution behavior. Fuzzers can use this information to manipulate the bytes that correspond to a part of the program's execution that it is interested in. VUZZER [19] is an early adopter of this technique, and proposes to mutate input bytes corresponding to a target comparison expression. Angora [6] improves upon this approach by using gradient descent to move the comparison operand to let the conditional jump move towards the other branch. Matryoshka [7] improves upon Angora's method by including the predicates of path constraints into the equation and solving them together. In contrast to using taint propagation, Redqueen [2] probes the bytes to identify which bytes correspond to which variable in the program.

Symbolic execution and concolic execution are another form of feedback. These methods generally collect symbolic path constraints and utilize a solver to produce satisfying inputs. KLEE [4] is a dynamic symbolic execution engine using the LLVM compiler framework. However, these methods frequently encounter problems such as path explosion and solver's inability to solve problems.

Research works like QSYM [31] combine concolic execution with greybox fuzzing to mitigate this issue.

State coverage is another field of research that attempts to model the program's execution behavior through tracing its state transitions. IJON and StateFuzz are two recent works that focus on state coverage, but focus more on recording the states of specific values. Finding such values require human input or specific program analysis, both are not easily scalable. In IJON's case, it requries manual labor in identifying state variables and their values, which is not applicable for large codebases such as the Linux kernel. For StateFuzz, it utilizes heavy static analysis, which requires heavy resources and may introduce false positives. In contrast, our approach uses a lightweight analysis routine that detects the definition and usage of static data within the kernel's binary to perform coverage feedback, resulting in better effectiveness, as shown in the previous section.

9 Conclusion

In this paper, we introduce the use of *Kernel Binary Coverage*, a novel coverage feedback mechanism for kernel fuzzers that increases the scope of coverage feedback to memory access patterns to the whole kernel binary. We propose a design that utilizes multiple mechanisms to be accurate and efficient, and in the process raises multiple design considerations and choices to balance efficiency and accuracy to satisfy the needs of kernel fuzzers. We implemented our approach as KBinCov and evaluated its performance, where statistics show that it outstrips Syzkaller and our own implementations of StateFuzz and IJON. In addition, we found 21 new bugs using KBinCov and Syzkaller in a fuzzing campaign.

Acknowledgements

We thank the anonymous reviewers for their insightful feedback. This research is sponsored in part by the National Key Research and Development Project (No.2022YFB3104000) and NSFC Program (No.92167101,62021002).

References

[1] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1597–1612, 2020.

[2] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, volume 19, pages 1–15, 2019.

[3] Alexander Bulekov, Bandan Das, Stefan Hajnoczi, and Manuel Egele. No grammar, no problem: Towards fuzzing the linux kernel without system-call descriptions. In *NDSS*, 2023.

[4] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[5] Libo Chen, Quanpu Cai, Zhenbang Ma, Yanhao Wang, Hong Hu, Minghang Shen, Yue Liu, Shanqing Guo, Haixin Duan, Kaida Jiang, et al. Sfuzz: Slice-based fuzzing for real-time operating systems. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 485–498, 2022.

[6] Peng Chen and Hao Chen. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725, 2018.

[7] Peng Chen, Jianzhong Liu, and Hao Chen. Matryoshka: fuzzing deeply nested branches. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 499–513, 2019.

[8] Stéphane Duverger and Anaïs Gantet. Gustave: Fuzz it like it's app. In *Proc. DMU Cyber Week*, pages 1–25, 2021.

[9] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[10] Google. Kernel address sanitizer. https://www.kernel.org/doc/html/latest/dev-tools/kasan.html.

[11] Google. Kernel concurrency sanitizer. https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html.

[12] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding Kernel Race Bugs through Fuzzing. In *IEEE Symposium on Security and Privacy*, pages 754–768. IEEE, 2019.

[13] Zhenpeng Lin, Yueqi Chen, Yuhang Wu, Dongliang Mu, Chensheng Yu, Xinyu Xing, and Kang Li. Grebe: Unveiling exploitation potential for linux kernel bugs. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2078–2095, 2022.

[14] Jianzhong Liu, Yuheng Shen, Yiru Xu, Hao Sun, and Yu Jiang. Horus: Accelerating kernel fuzzing through efficient host-vm memory access procedures. *ACM Trans. Softw. Eng. Methodol.*, 33(1), nov 2023.

[15] Zhengxiong Luo, Junze Yu, Feilong Zuo, Jianzhong Liu, Yu Jiang, Ting Chen, Abhik Roychoudhury, and Jiaguang Sun. Bleem: packet sequence oriented fuzzing for protocol implementations. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4481–4498, 2023.

[16] Zheyu Ma, Bodong Zhao, Letu Ren, Zheming Li, Siqi Ma, Xiapu Luo, and Chao Zhang. Printfuzz: fuzzing linux drivers via automated virtual device simulation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 404–416, 2022.

[17] Shankara Pailoor, Andrew Aday, and Suman Jana. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 729–743, Baltimore, MD, August 2018. USENIX Association.

[18] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 460–465. IEEE, 2020.

[19] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[20] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 167–182, Vancouver, BC, August 2017. USENIX Association.

[21] Yuheng Shen, Hao Sun, Yu Jiang, Heyuan Shi, Yixiao Yang, and Wanli Chang. Rtkaller: State-Aware Task Generation for RTOS Fuzzing. *ACM Trans. Embed. Comput. Syst.*, 20(5s), sep 2021.

[22] Yuheng Shen, Yiru Xu, Hao Sun, Jianzhong Liu, Zichen Xu, Aiguo Cui, Heyuan Shi, and Yu Jiang. Tardis: Coverage-guided embedded operating system fuzzing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(11):4563–4574, 2022.

[23] Zekun Shen, Ritik Roongta, and Brendan Dolan-Gavitt. Drifuzz: Harvesting bugs in device drivers from golden seeds. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1275–1290, 2022.

[24] Hao Sun, Yuheng Shen, Jianzhong Liu, Yiru Xu, and Yu Jiang. {KSG}: Augmenting kernel fuzzing with system call specification generation. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 351–366, 2022.

[25] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. *HEALER: Relation Learning Guided Kernel Fuzzing*, page 344–358. Association for Computing Machinery, New York, NY, USA, 2021.

[26] Dmitry Vyukov and Andrey Konovalov. Syzbot, 2015. https://syzkaller.appspot.com/upstream.

[27] Dmitry Vyukov and Andrey Konovalov. Syzkaller: an unsupervised coverage-guided kernel fuzzer, 2015. https://github.com/google/syzkaller.

[28] Mingzhe Wang, Jie Liang, Chijin Zhou, Yu Jiang, Rui Wang, Chengnian Sun, and Jiaguang Sun. RIFF: Reduced Instruction Footprint for Coverage-Guided Fuzzing. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 147–159. USENIX Association, July 2021.

[29] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data race fuzzing for kernel file systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1643–1660. IEEE, 2020.

[30] Y. Xu, H. Sun, J. Liu, Y. Shen, and Y. Jiang. Saturn: Host-gadget synergistic usb driver fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 51–51, Los Alamitos, CA, USA, may 2024. IEEE Computer Society.

[31] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 745–761, 2018.

[32] Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, Ming Yuan, Wenyu Zhu, Zhihong Tian, and Chao Zhang. StateFuzz: System Call-Based State-Aware linux driver fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3273–3289, Boston, MA, August 2022. USENIX Association.

[33] Wenjia Zhao, Kangjie Lu, Qiushi Wu, and Yong Qi. Semantic-informed driver fuzzing without both the hardware devices and the emulators. In *Network and Distributed Systems Security (NDSS) Symposium 2022*, 2022.

[34] Xiaochen Zou, Guoren Li, Weiteng Chen, Hang Zhang, and Zhiyun Qian. {SyzScope}: Revealing {High-Risk} security impacts of {Fuzzer-Exposed} bugs in linux kernel. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3201–3217, 2022.