# Finding Correctness Bugs in eBPF Verifier with Structured and Sanitized Program

Hao Sun
Tsinghua University

Yiru Xu
Tsinghua University

Jianzhong Liu
Tsinghua University

Yuheng Shen
Tsinghua University

Nan Guan
City University of Hong Kong

Yu Jiang
Tsinghua University

## Abstract

eBPF is an inspiring technique in Linux that allows user space processes to extend the kernel by dynamically injecting programs. However, it poses security issues, since the untrusted user code is now executed in the kernel space. eBPF utilizes a verifier to validate the safety of the provided programs, thus its correctness is of paramount importance as attackers may exploit vulnerabilities within it to inject malicious programs. Bug-finding tools like kernel fuzzers currently can detect memory bugs in eBPF system calls, but they experience difficulties in finding correctness bugs in the verifier, e.g., incorrect validations that allow the loading of unsafe programs. Because, unlike detecting memory bugs, where sanitizers can capture such errors once observed, automatically uncovering correctness bugs is very difficult, without an effective *test oracle* that determines if the verifier behaves correctly for given programs.

In this paper, we propose an effective approach to automatically detect the verifier's correctness bugs. Our core observation is that since the verifier aims to ensure that eBPF programs do not affect the security of the kernel, any illegal behaviors in verified programs are indicators of correctness bugs in the verifier. Indeed, we can convert the detection of logical errors in the verifier to traditional bug finding in eBPF programs. Based on such insight, we devise two indicators for correctness bugs and propose corresponding sanitation mechanisms to capture them, both of which naturally form an effective test oracle. We implemented our idea in a tool, namely BVF, which generates structured eBPF programs to pass the verifier, and subsequently, it finds correctness bugs by detecting runtime errors in verified programs with the indicators. Experiments show that although the verifier has received extensive scrutiny and has been intensively tested by tools like Syzkaller and Buzzer, BVF still found 11 previously unknown vulnerabilities in eBPF, of which six are correctness bugs of critical severity in the verifier.

***CCS Concepts:*** • **Security and privacy** → *Operating systems security*.

***Keywords:*** Testing, OS Kernel, eBPF Verifier

**ACM Reference Format:**
Hao Sun, Yiru Xu, Jianzhong Liu, Yuheng Shen, Nan Guan, and Yu Jiang. 2024. Finding Correctness Bugs in eBPF Verifier with Structured and Sanitized Program. In *European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3627703.3629562

## 1 Introduction

eBPF [14] is a kernel extension technology that supports the injection of user-written programs into almost every kernel module. Utilizing such programmability, the user space processes can achieve various goals and extend the kernel at runtime. Currently, eBPF is widely adopted by industry and academia [27]. Data centers, for instance, use eBPF to perform efficient packet filtering with far better performance than iptables-based mechanisms [13, 39, 48]; some researchers have used eBPF to implement kernel probing [37, 50] and security monitoring [12]; and most recently, eBPF has been integrated to optimize the scheduler in Linux [20], where new scheduling policies can be expressed as eBPF programs, thus achieving flexible scaling.

To ensure that eBPF programs from user space do not affect the security and stability of the operating system, the kernel utilizes a verifier to check the program's integrity, e.g., validating that the program only accesses restricted memory in a legal way. Specifically, the eBPF verifier performs a sophisticated analysis, consisting of more than 27,000 lines of C code, on the programs to collect relevant states and verify the correctness of sensitive operations before loading them. Hence, the verifier has become the most complex and error-prone component in the eBPF subsystem. Such complexity introduces a diverse set of vulnerabilities [1, 2],

*Yu Jiang is the corresponding author of this paper.


*EuroSys '24, April 22–25, 2024, Athens, Greece*

ACM ISBN 979-8-4007-0437-6/24/04…$15.00
https://doi.org/10.1145/3627703.3629562

which are attractive to attackers given that eBPF has provided code execution ability in the kernel. Exploiting such vulnerabilities, attackers are likely to inject malicious programs, perform arbitrary accesses, and achieve local privilege escalation (LPE),. Take CVE-2022-23222 [3] for instance. As shown in Listing 1, the verifier incorrectly allows ALU on nullable pointers, resulting in out-of-bound access. The vulnerability can be further exploited to achieve LPE, which is demonstrated with a working proof-of-concept [4]. Therefore, detecting and fixing the verifier's correctness bugs is of paramount importance for the entire kernel's security.

```
0: r1 = map_fd0
1: call map_lookup_elem
2: r8 = r0          ; R8=map_value(ks=4, vs=4096)
3: r1 = map_fd1
4: r2 = 8192
5: call ringbuf_reserve
6: r1 = r0
7: r1 += 1          ; ALU on nullable pointer here
                    ; Verifier believes r0 = 0 and r1 = 0
                    ; However, r1 = 1 at runtime.
8: r1 *= -1024
9: r8 += r1
10: r0 = *(u64 *)(r8 + 0) ; Out-of-bounds access here
```

**Listing 1.** CVE-2022-23222. The program (simplified) triggers an out-of-bounds access after loaded, due to an improper checking that allows arithmetic operations on nullable pointers. After #2 and #6, the verifier marks R8 as a pointer to the map value and marks R1 as a nullable pointer. At #8, the verifier believes R1 equals zero, which is incorrect due to #7, leading to the invalid access at #10.

Existing works on security testing of the eBPF verifier mainly focus on manual testing and kernel fuzzing. eBPF maintainers have built a test engine [6] for the verifier that supports automatic test case loading and results checking. The kernel developers have created a large number of self-tests, which contain eBPF programs that cover a variety of scenarios and can effectively test if the verifier works as expected. However, manually-written tests are incapable of keeping up with the development of the verifier, and numerous corner cases and their combinations are not yet covered. Indeed, despite that eBPF is one of the most sufficiently tested components in the kernel, security vulnerabilities within which continue to surface. As one of the most effective bug-finding approaches, kernel fuzzing has also been adopted for eBPF testing. For instance, Syzkaller [43] has been integrated into eBPF upstream and has proven to be effective in detecting memory errors in eBPF system calls. Nevertheless, the eBPF programs generated by kernel fuzzers are easily rejected by the complicated checks of the verifier, and they can hardly reach and discover correctness bugs, prompting the need for a better approach towards correctness bug finding in the verifier.

However, automatically uncovering correctness bugs in the verifier is very difficult without effective approaches. Specifically, in traditional dynamic testing or fuzzing, erroneous behaviors in a software system can be triggered by generated inputs and captured by inserting runtime checks on certain operations. For instance, fuzzers can automatically detect memory bugs with the assistance of existing sanitizers, which hook relevant load/store instructions to collect memory states and check whether the access is within bounds at runtime. The aforementioned automatic check is feasible mainly because the property to check is relatively trivial, i.e., essentially the memory sanitizer inserts `assert(valid-access)` statements. However, in the case of correctness bug finding in the verifier, even though fuzzers can randomly generate eBPF programs, devising runtime checks to capture correctness bugs is hard, because the verifier performs comprehensive validations on eBPF programs to check a huge set of properties. Automatically checking correctness, i.e., `assert(valid-analysis)`, requires inserting extensive assertions into various locations of the verifier, thus demanding domain knowledge of the verifier and intensive manual efforts. For instance, Agni [41] generates verification conditions of the verifier from source code and utilizes SMT solvers for checking, successfully advancing the verifier's correctness. However, its approach mainly considers range tracking, a relatively small portion, which would require further efforts to extend. Another relevant work CSmith [47] generates undefined-behavior-free programs and performs differential testing for compiler bugs. However, applying such an approach is hard as it requires reference implementations that are as mutual and precise as the current verifier in Linux.

Therefore, a key challenge for correctness bug finding is to devise an effective *test oracle* [21] that automatically determines if the verifier's judgment is correct for a given program. Since directly determining the correctness is hard as demonstrated, we adopt an alternative view: in essence, correctness bugs in the verifier are errors that incorrectly load eBPF programs capable of affecting kernel stability, thus we can convert such detection into bug finding in verified programs. Specifically, erroneous eBPF programs loaded into the kernel may lead to invalid kernel states in two fundamental ways: either through executing invalid load/store instructions in the program or by executing kernel routines invoked by the program indirectly. The implication is that if we can trigger and capture either of the two abnormal behaviors in verified programs, then we have found a correctness bug. This view is beneficial because we now do not need to directly check each possible kind of incorrect behavior in the verifier, which would otherwise be extremely complex if not even possible. Instead, the verifier's correctness bugs are eventually reflected as two kinds of abnormal behavior in eBPF programs. For instance, the improper check in CVE-2022-23222 (Listing 1) collected incorrect registers' states

propagated to the following validation, thus eventually appearing as an out-of-bounds access. These behaviors act as two indicators for correctness bugs, and we capture them with sanitation mechanisms. Consequently, the indicators and sanitation naturally form an effective test oracle.

We implemented our idea in a tool, namely BVF, which utilizes the following steps to detect correctness bugs in the verifier. First, in order to trigger the indicators, BVF needs to generate complex eBPF programs while passing the verifier efficiently, for which, we propose a lightweight structure that partitions programs into multiple fundamental sections, thereby guiding input synthesis. Second, to capture the indicators, we utilize a dispatch-based sanitation and perform instrumentation upon verified programs, which is conducted entirely at the eBPF instruction level and thus is efficient and architecture-independent. Finally, BVF detects correctness bugs by continuously executing and triggering bugs in sanitized programs. Consequently, we discovered 11 previously unknown vulnerabilities in the eBPF subsystem, of which six are the verifier's correctness bugs. Although Syzkaller and Buzzer have been testing eBPF with extensive computing resources and the verifier has been checked by numerous manual tests, those six bugs have never been detected before. Most of the uncovered correctness bugs are critical, which can result in out-of-bounds access, deadlock, kernel panic, etc. All the vulnerabilities have been confirmed and fixed by corresponding patches proposed by maintainers and us. This demonstrates that our technique is highly effective in finding correctness bugs. Our key contributions are as follows:

- Compared to existing works, which mainly target memory bugs in eBPF, we propose an effective test oracle for correctness bug detection in the verifier. The oracle is formed by two indicators and a corresponding sanitation mechanism and is practical and effective for capturing a wide range of correctness bugs.
- To effectively test the verifier and trigger the test oracle, we propose a lightweight structure to guide the generation of the eBPF program. Such a technique is capable of synthesizing interesting eBPF programs and improving the success rate of passing the verifier.
- We implemented our approach in BVF, and we have discovered and reported 11 previously unknown vulnerabilities in eBPF, of which six are correctness bugs in the verifier of critical severity.

## 2 Background

**eBPF Subsystem.** Extended Berkeley Packet Filter (eBPF) is a kernel extension technique that provides a RISC-like instruction set [5], program verifier, just-in-time compiling engine (JIT) [30], and execution environment inside the Linux kernel. The instruction set of eBPF is relatively minimal, consisting of mainly four types of instructions: load, store, jump, ALU, and related variants; however, user space processes can build feature-rich programs with eBPF by constructing instruction sequences directly or using a front language, such as C language. The verifier, the most complicated component of eBPF, validates the security of the program. Once the program passes the verification, eBPF utilizes the internal JIT engine to compile it to native code for execution. Although an interpreter is available, in practice, JIT is the default option that many important features require, e.g., calling kernel functions, and the interpreter is disabled for security concerns. eBPF also provides an execution environment, including mount points, helper functions, data structures accessible for the programs, etc. For instance, eBPF programs can communicate with the kernel via helper functions or a limited set of kernel functions, and the eBPF map allows programs to interact with user space processes. Finally, the JITed programs can be mounted to various places in the kernel, and almost every kernel location can be extended when used in conjunction with the kprobe [25] mechanism.

**Table 1.** Example of the verifier's workflow. The verifier tracks the state of each register (R0 ~ R10), where R1 is a pointer to context, R10 is a pointer to stack, and other registers are not initialized at the start. The first instruction loads the address of the map to R1, and the verifier changes its state to a pointer to the map and records corresponding information, e.g., key size. The following three instructions store value to the stack; the verifier requires that all the memory must be properly initialized before use. With R1 storing a map pointer and R2 containing a pointer to a key value located on the stack, the last instruction invokes the helper call to look up the value in the map, after which R0 stores a nullable pointer to the map value.

| BPF Insns | Description | Register State |
|---|---|---|
| func entry | *initial state of regs* | R0 = not_init |
| r1 = map_fd | *load map fd to R1* | R1 = map_ptr (ks=8...) |
| r2 = r10 | *mov stack pointer (fp) to R2* | R2 = ptr_to_stack |
| r2 += -8 | *add offset -8 to R2* | R2 = ptr_to_stack (off=-8) |
| *(u64 *) r2 = 0 | *store 0 (eight bytes) to stack* | fp-8 = 0 |
| call map_lookup_elem | *call bpf map helper func* | R0 = map_value_or_null |

**eBPF Verifier.** The eBPF verifier [7, 17, 40] utilizes static analysis to conduct validation on the safety of the provided programs. Specifically, it models all the possible register states in the abstract domain and collects program information by simulating the execution of each instruction in different paths. The verifier checks the correctness of each instruction based on the collected states, and programs containing illegal operations, such as using uninitialized registers and out-of-bounds accesses, are rejected. For instance, after simulating the execution of a map file descriptor loading instruction, the verifier marks the corresponding register's state to `CONST_PTR_TO_MAP` and validates the following operations on the register with relevant information of the map. At a high level, the state of each register is classified

into three categories: uninitialized, scalar value, and pointer. Registers are uninitialized before any valid loading, and for non-pointer values, the verifier uses scalar values to represent accurate ranges. The objects to which registers may point are also detailed modeled. For instance, the verifier currently supports more than ten types of pointers, e.g., map pointer, packet pointer, kernel data structure pointer, etc. Table 1 demonstrates a simple workflow of the verifier.

The correctness of the eBPF verifier is essential because bugs in it could potentially result in malicious programs being loaded into the kernel. Such abnormal behaviors are less difficult to exploit than vulnerabilities in other subsystems given that the attackers have already gained the ability, although restricted, to execute code in the kernel, exploiting which further arbitrary accesses have a high probability of being achieved. While some Linux desktop distributions choose to disable unprivileged eBPF, many data centers/servers utilize unprivileged eBPF to conduct more strict checks, e.g., the verifier enforces more restrictions on programs loaded by unprivileged users. From a kernel development perspective, maintainers do not make any assumption about user space, i.e., regardless of the users of eBPF, any verifier's correctness bugs can potentially lead to erroneous programs being loaded, affecting kernel stability. Therefore, detecting and fixing correctness bugs in the verifier is of great importance for kernel security.

**Verifier Testing.** Existing works on eBPF verifier testing mainly focus on manual testing and fuzzing. Specifically, eBPF maintainers utilize a test engine for the verifier to automatically execute test cases and check results. They have also created a large amount of eBPF programs covering a variety of use cases to effectively test the verifier's algorithm. However, manual testing is less scalable compared to the rapid changes that occurred in the eBPF subsystem, and therefore we mainly focus on automated test approaches.

Fuzzing is a promising vulnerability detection technique [9, 28, 51]. Its idea is to generate or mutate inputs to trigger abnormal program behaviors and observe such anomalies with the assistance of sanitizers [34–36]. Fuzzing has also been adopted in assisting kernel testing, where the primary workflow is similar to that in user space fuzzing [15, 16, 22, 29, 45, 49], but each step is optimized for kernel scenarios [26, 31, 33, 37, 38, 46]. To automatically test the kernel, the fuzzer generates system call sequences, where the input structures are carefully constructed to pass the basic parameters validation of the kernel. After invoking the system calls, the execution feedback, e.g., code coverage [8], is collected, which is then consumed by the fuzzer to improve its effectiveness. Take Syzkaller [43] (syzbot) as an example, it generates system call sequences based on predefined system call descriptions [44] and has already found thousands of bugs in the Linux kernel with the assistance of many kernel sanitizers [18, 19].

However, existing testing tools experience difficulties in the verifier's correctness bug finding. For instance, while Syzkaller can test eBPF by randomly generating eBPF instructions, its approach mainly targets memory bugs during the execution of system calls but does not perform any correctness checks or utilize any test oracles. We will discuss more about related works in Section 7.

## 3 Correctness Bug Indicators

In this work, we intend to detect the verifier's correctness bugs by utilizing bugs in eBPF programs as indicators and capturing them with effective mechanisms. The ensuing problems are 1) what kinds of bugs in eBPF programs need to be considered as indicators for correctness bugs; 2) how to trigger and capture those indicators during runtime.

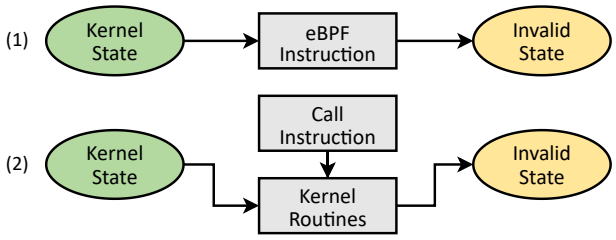


**Figure 1.** Two kinds of abnormal behaviors are two major indicators for correctness bugs in the verifier. The first is that eBPF programs affect the kernel directly when executing their instructions in an illegal way. The second is when programs execute kernel routines after invoking the call instructions in an unexpected manner.

For the first problem, we need to analyze the intrinsic behaviors of eBPF programs. In essence, erroneous programs loaded into the kernel due to correctness bugs in the verifier can cause invalid kernel states in *two fundamental ways* as depicted in Figure 1. The first is that programs *directly* affect the kernel's correctness when executing their instructions. For instance, programs with out-of-bounds accesses can lead to invalid kernel states after executing the corresponding memory access instructions. The second way is when programs execute kernel routines after the call instructions in an unexpected manner that *indirectly* impacts the kernel. For example, a program may cause kernel deadlock during running in the kernel functions, despite that the instructions of the program do not affect the kernel states directly. In principle, the majority kinds of correctness bugs in the verifier are eventually appeared and reflected as those two kinds of abnormal behaviors in eBPF programs. Therefore, we can utilize those two types of illegal behaviors in eBPF programs as two effective indicators for correctness bug finding.

To efficiently trigger the indicators for correctness bug finding, BVF should be capable of generating complicated programs that can pass the verifier. To accomplish this, the

generated programs must satisfy various basic properties, e.g., ensuring all the generated instructions are correctly encoded and registers are initialized before being used, to bypass early validations. BVF should also be capable of generating complex behaviors, such as function calls and nested jumps. We observe that eBPF programs have certain construction patterns, indicating that we can divide programs into fundamental sections with different kinds of behaviors, and construct complex programs by combining them. We defer to Section 4.1 for a detailed description.

To capture the indicators during runtime, different mechanisms are required, and we demonstrate this in the following explanation of the indicators.

### 3.1 Indicator#1

The scope of the first indicator mentioned above involves each kind of instruction in eBPF programs, and we can refine it by further analyzing the semantics of eBPF instruction.

**Indicator Definition.** As mentioned in Section 2, the eBPF instruction set is minimal by design, containing mainly four types of instructions. However, not all of them can impact the kernel. Specifically, 1) the arithmetic operations cannot lead to kernel errors directly because they do not access the kernel states, meaning the side effects are limited within the programs; 2) the `offset` operand of jump instructions in eBPF can only be a constant value, indicating that each jump in a verified program can only be performed within the program's boundary, and such an instruction can not impact the kernel either. Therefore, the guilty instructions are mainly load/store operations. Ideally, the verifier performs sophisticated analysis to collect the registers' states and validate the access, thus ensuring the programs' memory safety. Practically, the verifier may make mistakes in different analysis phases and propagate inaccurate information to the memory access validation. In fact, the first step an attacker takes when exploiting correctness bugs is to construct illegal memory access. The insight here is that load/store are the major sources for the first kind of abnormal behavior, i.e., invalid load/store is the indicator#1 for correctness bugs.

In order to effectively capture the indicator#1, we need to perform runtime checks on these instructions, specifically validating memory accesses in eBPF programs.

**Indicator Capture.** To capture invalid load/store instructions, we need to perform memory sanitation in eBPF programs. Although Linux has Kernel Address Sanitizer (KASAN) to validate memory access in itself, such a mechanism can not be applied in eBPF programs directly. Specifically, in order to perform memory checking, KASAN first instruments sensitive operations in the kernel and records metadata of all the allocated memory in a separated region named shadow memory, thus the instrumentation and the shadow memory are prerequisites for memory sanitation. However, eBPF programs passing the verifier are compiled to native code at runtime by the eBPF JIT engine without any instrumentation, indicating that KASAN is incapable of detecting memory bugs in them. Nevertheless, we observe that all the memory accessible to eBPF programs is either preallocated before the execution or constructed by the kernel functions, implying that KASAN has already recorded that memory into shadow memory. Consequently, to capture invalid memory access in eBPF programs, we can dispatch necessary load/store instructions to kernel functions that have been instrumented for memory sanitation to achieve indirect checking. Furthermore, the dispatch can be performed by instrumenting the programs passing the verifier entirely at the eBPF instruction level, thus being efficient and architecture-independent. We illustrate our memory sanitation in Section 4.2.

```
0: r1 = map_fd           ; R1=map_ptr(ks=4, vs=4)
1: r6 = *(u64 *)(r1 + 8) ; R6=bpf_map->inner_map_data
                         ; PTR_TO_BTF_ID, null at runtime
2: r2 = r10
3: r2 += -4
4: *(u32 *)(r2 + 0) = 0
5: call bpf_map_lookup_elem ; R0=map_value_or_null
6: if r6 != r0 goto pc+1 ; Incorrectly mark R0 as non-null
7: r0 = *(u32 *)(r0 + 0) ; FLAW: trigger runtime check here
8: exit
```

**Listing 2.** The program constructed by BVF triggers invalid memory access due to a correctness bug in the verifier. After #1, the verifier marks R6 as PTR_TO_BTF_ID, which is a pointer to a kernel object that programs do not need to perform a null check. After #5, R0 is a nullable pointer to the map value. During validating #6, the verifier incorrectly marks R0 as non-null due to a correctness bug, leading to the following invalid memory access. BVF can discover this vulnerability because the generated program triggered our runtime checks.

**Correctness Bug Example.** As shown in Listing 2, an eBPF program constructed by BVF can trigger invalid memory access during execution after passing the verifier. The program is an indicator of a verifier's correctness bug because its load can affect the kernel and can potentially be exploited by the attackers. The root cause of this correctness bug is due to the incorrect nullness propagation analysis, as explained in Section 6.2. Such an analysis pass collected incorrect register states, which are propagated to the following memory access validation, resulting in the loading of such a memory-unsafe program. In addition, BVF is also capable of uncovering CVE-2022-23222 by generating an eBPF program that contains out-of-bounds access to the eBPF map as shown in Listing 1. The aforementioned two programs, both containing invalid load/store instructions, effectively reflect two different correctness bugs in the verifier, and BVF can detect these illegal accesses by leveraging our sanitation. This demonstrates the effectiveness of indicator#1 and our sanitation for reflecting and capturing various correctness

bugs; otherwise, it would be difficult to detect such bugs by simply executing native code generated by the JIT engine.

### 3.2 Indicator#2

Indicator#1 reflects bugs caused by eBPF programs directly, while indicator#2 represents bugs caused by loaded programs indirectly. Both indicators complement each other, and correctness bugs eventually appear as one of them.

**Indicator Definition.** eBPF supports function invocation with the `call` instruction (a special kind of jump operation) and provides programs with hundreds of helper functions, thus enabling flexible interaction between the programs and the kernel. However, the flexibility presents corresponding complexity to the verifier. The verifier needs to ensure that eBPF programs invoke the helpers in a correct manner by sufficiently checking if the input states in the programs match the requirements of the helpers' prototypes. In practice, the verifier may incorrectly reason about registers' states or analyze the safety of certain helpers' invocation; thus the programs may break the kernel during the execution of helper functions. Consequently, bugs caused during kernel routines' execution invoked by loaded eBPF programs are the indicator#2 for correctness bugs and the major reason for such a bug is that eBPF programs invoke kernel routines with unexpected inputs that the verifier cannot detect. Indicator#2 in conjunction with indicator#1 covers eBPF programs that impact the kernel *directly* and *indirectly*, thus constituting a wide range of correctness bugs.

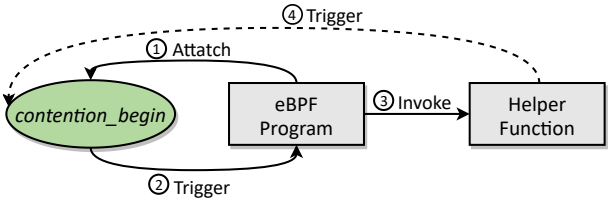


**Figure 2.** Another correctness bug found by BVF allows programs that may lead to kernel deadlock to be loaded. The program for this vulnerability is attached to the tracepoint `contention_begin`, and then the program is triggered whenever the tracepoint is reached. After calling the helper function that attempts to acquire a lock, the tracepoint and the eBPF program are triggered again, leading to the recursion and inconsistent lock state kernel errors.

**Indicator Capture.** In order to capture indicator#2, we need to uncover bugs that occurred during the kernel routines' execution. The scope of such bugs is huge, including memory errors, data racing, etc., and different mechanisms are required to capture them correspondingly. However, unlike eBPF programs, which are compiled after loading by the JIT engine, the routines that the programs can invoke are part of the kernel, i.e., they are compiled alongside the rest of the kernel code, not at runtime. Consequently, different from indicator#1, we can adopt self-check mechanisms in the kernel to capture indicator#2 and existing mechanisms can catch the majority of runtime bugs in these routines. For instance, since the routines are compiled with the kernel, KASAN has already instrumented their code for state collection and sanitation and the runtime locking correctness validator [11] in Linux is also capable of detecting data races in those routines. To efficiently utilize indicator#2 for correctness bug finding, BVF should be capable of generating complicated programs that can pass the verifier effectively.

**Correctness Bug Example.** Take another correctness bug found by BVF for example. BVF synthesized an eBPF program, and its program type is kprobe. As shown in Figure 2, the program calls an eBPF helper function that attempts to acquire a local lock during execution. When the program is attached to the tracepoint `contention_begin`, a deadlock would potentially occur. This is because the program is triggered once the tracepoint is reached, after which the helper function invoked by the program would trigger the tracepoint again due to the acquisition of the lock. This, in turn, triggers the program again and leads to the recursion and inconsistent lock state errors, thereby revealing a correctness bug in the verifier. We can find this vulnerability because the generated program can pass the verification and trigger indicator#2, which can be effectively captured by runtime locking correctness validator in Linux. For other bugs that may be triggered during the execution of kernel routines invoked by eBPF programs, we can trigger them by continuously generating interesting programs and capturing them with different kernel mechanisms.

## 4 BVF Design

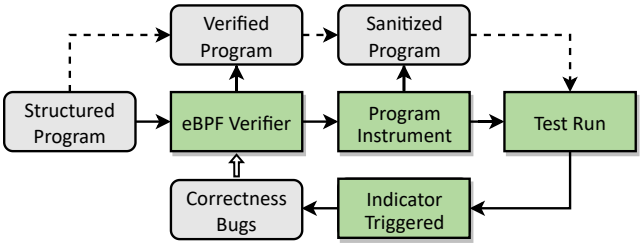


**Figure 3.** Overall workflow. First, BVF generates eBPF programs with a structured design. The generated programs are forwarded to the verifier for checking, during which the verifier rewrites the programs for various purposes. BVF performs instrumentation for memory accessing checking at runtime on verified programs at the end of the rewriting phase. Finally, BVF detects correctness bugs in the verifier by triggering bugs in the sanitized programs.

We implemented our idea in BVF and Figure 3 illustrates its overall workflow. In order to generate complicated eBPF

programs while passing the verifier to trigger the indicators, BVF utilizes a lightweight structured design to guide the program synthesis. The generated program is then forwarded to the verifier by BVF via the `bpf()` system call. The verifier conducts the complete validation of the program. To effectively capture indicator#1 in the program, BVF performs instrumentation on the verified program at the end of eBPF's rewriting phase. The instrumentation dispatches all the necessary load/store operations in the programs to the functions we proposed in the kernel that have already been instrumented by KASAN, thus achieving memory access sanitation in eBPF programs. Finally, the sanitized program can be used to detect correctness bugs in the verifier. BVF performs test runs on the loaded program, and a correctness bug is discovered once the indicators are triggered.

### 4.1 Structured Program Generation

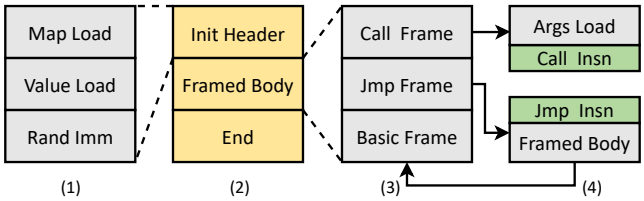


**Figure 4.** Program structure. At the top level, each eBPF program is partitioned into three parts: the init header, the framed body, and the end section. The init header selects a set of loading instructions, e.g., map loading and random value loading, to initialize registers. The framed body further divides the programs and contains a set of lower-level sections: the call frame, the jump frame, and the basic frame. This enables BVF to generate complex programs.

In order to trigger the two indicators efficiently, BVF should be capable of generating interesting eBPF programs that can pass the verifier. The insight of BVF's program generation is that we can partition eBPF programs into several fundamental sections, and sophisticated behaviors can be composed by combining the sections. In BVF, except to ensure the validity of the instructions, we utilize a structured design as shown in Figure 4 to guide the generation of the program. The structure partitions an eBPF program into three sections at the top level as shown in (2) of the figure: the init header, the framed body, and the end section. The init and end sections assist the generated programs in passing early validations in the verifier, thus allowing interesting behaviors contained in the framed body to be verified and loaded. Specifically, as aforementioned, eBPF requires that all the registers need to be initialized before any accessing and the program must contain valid exit instructions. The init header and end section enable the generated programs to satisfy these basic constraints. Overall, the program generation follows the proposed structure. BVF first emits instructions in the init header and end frame and then keeps selecting one of the frame kinds in the main body with equal probability and emits instructions in the selected frame accordingly. The following are the details of each top-level section:

- **Init Header:** performing initialization of registers by selecting register value loading instructions.
- **Framed Body:** the major part of the eBPF program, and further contains a set of lower-level sections, supporting complex program generation.
- **End Section:** this section is used for proper program ending and currently contains valid exit instructions.

Except for passing the basic checks, the init header also sets registers to various interesting initial states, thereby facilitating the construction of the complex operations on them. Specifically, as shown in (1) of the figure, the candidates for loading instructions in this section are all the possible objects that the programs can access, e.g., map file descriptors, map value, BTF file descriptors, and random 64-bit immediate. BVF constructs the corresponding resources in the kernel before execution if the generated programs intend to load and operate on them. In addition, registers used for parameter passing are skipped in this section because they already have complex states, e.g., R1 in eBPF is initialized as a pointer to the context data.

We observe that the behaviors of eBPF programs can be essentially classified into three categories: operations on the accessible objects that occur inside the program, interactions with the kernel achieved by call instructions, and selections of these actions through jumps, implying that we can break down the framed body to three intrinsic components correspondingly. Specifically, we utilize a linear structure as shown in (3) in Figure 4 to represent programs given that only bounded loops are allowed in eBPF. All four types of instructions are classified into three main sections: the basic frame, the jump frame, and the call frame. The framed body contains a set of those frames. Instructions that do not cause changes in the control flow of the programs are arranged in the basic section, and various operations on kernel objects or accessible memory are combined in this section. The call instruction is a special kind of jump that enables eBPF programs to communicate with outside, and it is capable of triggering complicated logic in the verifier. Hence, we utilize the call frame to devise such a behavior. Finally, since the jump instruction can effectively demarcate the program state, the jump frame is adopted to achieve this effect and ensure the validity of the jump target. The following are the details of each section:

- **Call Frame:** contains loading instructions used for setting the value of R1 to R5, which are the registers used for parameters passing; the target of the call can be helper functions, kernel functions, and pseudo eBPF functions.

- **Jump Frame:** starts or ends with a jump instruction, the body contains multiple other frames; the offset of the jump is the number of instructions in the body, thus ensuring the correctness of the control flow.
- **Basic Frame:** contains instructions that do not affect the control flow of eBPF programs, all the load-/store/ALU instructions, and special operations, e.g., atomic arithmetic, are in this section.

In the basic frame, various basic operation patterns on all the accessible objects are generated. We achieve this by first recording the registers' states in different program points, and then synthesizing operations according to the states. For instance, for registers pointing to maps or context, we generate direct map value updating or map access, and context accessing, thus a variety of different behaviors are formed. In the loading part of the call frame as shown in (4) in Figure 4, the states required for interaction with the kernel or user space are filled into parameter passing registers (R1 to R5), and the call instruction is subsequently issued. eBPF supports hundreds of helper functions and many kernel functions, with the call frame, BVF can explore them continuously during the testing campaign. Furthermore, for back-edge jumps, i.e., the offset operand of the jump instruction is negative, we restrict the two operand types to register and constant and utilize a loop block variable that changes per iteration in the jump condition and bound it with an immediate value, thus reducing the occurrence of unbounded loops, i.e., loops with infinite iterations. BVF is capable of devising interesting control flow behaviors, e.g., nested jumps, when used in conjunction with other frames in the jump body, and simulating unrolled loops by utilizing the fuzzer's mutation operations to duplicate adjacent instructions.

### 4.2 Memory Access Sanitation

To capture the triggered indicators for correctness bug finding, we need to sanitize the load/store instructions in the generated programs. Specifically, eBPF programs are allowed to access a fixed-size stack, context states, kernel objects, and eBPF data structures, and the verifier conducts complicated validations upon these operations. Hence, if illegal access to those states, e.g., out-of-bounds write, occurs and is captured during execution, then, corresponding correctness bugs are uncovered. Although KASAN is incapable of performing sanitation on eBPF programs as illustrated in Section 3, we observe that the metadata corresponding to all the states mentioned are well recorded in the shadow memory. For instance, the context and the stack of eBPF programs are pre-allocated before execution, and all other objects are constructed by kernel routines that are already instrumented by KASAN. Based on the above observation, we can accomplish memory sanitation for eBPF programs effectively.

The workflow of our mechanism is as follows. First, the target address and the size of the memory that load/store instructions access are intercepted, and such information is dispatched to the sanitizing functions via the eBPF call instruction, i.e., the dispatching can be realized entirely at the eBPF instruction level. The sanitizing functions are located in the kernel and are instrumented with the KASAN sanitation during kernel compilation. The validity of the memory access is checked by comparing the target address with the information recorded in the shadow memory when the sanitizing functions are invoked, thereby achieving memory access sanitation in eBPF programs. This procedure is performed when the verifier rewrites the programs that pass the checks.

Figure 5 illustrates the instrumented eBPF instructions for sanitizing the eight-byte loading. First, since R1 is used in eBPF for passing the first parameter of functions, we back up the original state of R1 and load the target address of the memory access into it, and an auxiliary register named R11 in eBPF that is only visible internally (only R0 to R10 are visible to the program) is adopted for the backup. For other registers, e.g., R0 is overwritten for the function return value, we back up their states into an extended stack space that is also invisible to the program. Then, BVF selects the sanitizing functions based on the accessing size and mode, e.g., a double word size access corresponds to a function called `bpf_asan_load64()`. Finally, the accessing check is triggered when the sanitizing function performs the actual load/store, after which the registers' states are restored and the original instruction is appended.

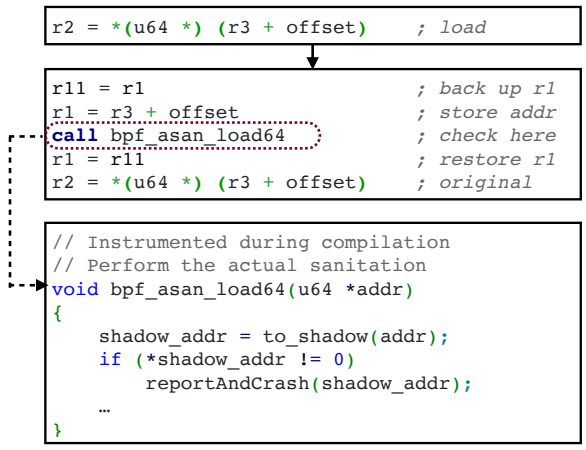


**Figure 5.** Instrumentation for eight bytes sized memory accessing. First, since R1 is adopted for parameter passing, the original state of R1 is backed up to an auxiliary register named R11, and the target address is stored in R1. Then, the call instruction that invokes the sanitizing function corresponding to the access size and mode is emitted, and the function performs the actual sanitation. Finally, R1 is restored and the original instruction is emitted.

Besides, BVF also instrument checks on some sensitive ALU operations invoked between a pointer and a scalar based on the verifier's knowledge. Specifically, the verifier classifies ALU operations based on the value types of the involved registers. For arithmetic instructions conducted on a pointer and a scalar, the verifier calculates a limit value called `alu_limit` to verify that the scalar contained in the register is a legal offset to the pointer. The `alu_limit` is a relative value calculated based on the ALU kinds and the sign of the offset operand, i.e., for `r0 += offset`, the `alu_limit` would be the lower bound if the offset is negative and vice versa. For ALU instructions with this information, BVF emits instructions in the sanitized program to check if the value of the register at runtime is within the `alu_limit` as the verifier expects; otherwise, an accessing error is reported. Essentially, the functionality of the instrumented instructions is equivalent to the assertion expression: `assert(offset < alu_limit)`.

In addition, we utilize the following strategy to reduce the amount of instructions injected. First, the load/store instructions that use R10 as the source or target register are skipped. Specifically, R10 stores the stack pointer and is read-only, which means that any program that attempts to modify it will be rejected. Since the `offset` operand in load/store instructions is a constant, the target address `R10 + offset` can be calculated during verification. For instance, for the load instruction `r0 = *(u64 *) (r10 + -8)`, the target address `(R10 + -8)` is a constant value known during verification. Since the stack size of eBPF programs is fixed at 512 bytes, the correctness of such load/store instructions can be validated by comparing the two constants: the target address and the stack size, thus we do not instrument these accesses. Second, instructions emitted by other rewrite passes in the verifier are omitted. Specifically, other rewrite passes may transform some load/store instructions into multiple ones, e.g., direct packet access instructions, and we only instrument the original instruction once to reduce the footprints.

## 5 Implementation

The implementation involves two major parts: the fuzzer and the modifications to the Linux kernel. We implement the aforementioned structured program generation by modifying Syzkaller. Specifically, our program generation routine consists of the following. We leverage system call generation techniques in the fuzzer. The top-level structures of the eBPF programs are represented in the system call descriptions [44], where the init header, the framed body, and the end frame are in type definitions. Possible instructions are represented as specific types. For other complex structures, we leave dummy structure definitions and fill them by extending the fuzzer's generation with custom generators and mutators.

The memory sanitation of eBPF programs is implemented in three patches for the Linux kernel. The first patch introduces the sanitizing functions and the instrumentation of store instructions in the kernel. All the sanitizing functions named `bpf_asan_store()` are added and the whole instrumentation is conducted in the `bpf_misc_fixup()` phase in conjunction with other rewrites passes so that no additional ad-hoc phase is required. Furthermore, the first patch also filters and detects instructions emitted by other rewrite passes to reduce the amount of instrumented instructions in the fixup phase. The second patch presents the sanitation of load instructions and the corresponding checking functions named `bpf_asan_load()`, similar to the first one. The last patch modifies the existing rewrite of `alu_limit` by adding runtime checks when such information is available and sanitation is enabled. The above modifications can be turned on with `Kconfig` in the case that KASAN is also available.

In addition, we follow the design of Syzkaller, the state-of-the-art kernel fuzzer, to best utilize its testing capability. For instance, we reuse its feedback mechanism, which collects coverage information and comparison operations, but we only instrument eBPF during compilation. The coverage information enables BVF to preserve interesting eBPF programs triggering new verifier's checking behaviors so that the following generation can base on the saved programs, thereby exploring the verifier iteratively.

## 6 Experiment

In this section, we evaluate the correctness bug-finding capability of BVF on recent versions of Linux by deploying BVF to conduct verifier testing for two weeks. At the same time, we also deploy and compare the bug-finding capability of BVF with that of Syzkaller and Buzzer to demonstrate the effectiveness of the proposed indicators. We chose Syzkaller because it is the state-of-the-art kernel fuzzer that has been integrated into the eBPF upstream testing and Buzzer is also a representative eBPF fuzzer. In addition, we also evaluate the effectiveness of the proposed program structure and the overhead of the sanitation mechanism. For the former, since the structure is used to cover more verifier's code, we compared the testing performance of BVF to that of Syzkaller and Buzzer. To evaluate the overhead of sanitation, we utilized manually-written test cases in eBPF self-tests as datasets to calculate the execution time and instruction footprints before and after sanitation. In summary, we design experiments to address the following questions:

- **RQ1:** Can BVF uncover previously unknown correctness bugs in the verifier?
- **RQ2:** How effective is the program structure in improving the bug-finding capability of BVF?
- **RQ3:** How much overhead does sanitation present?

**Experiment Setup.** We describe the setup adopted in the following experiments. All the experiments were conducted on a Linux server with a 40-core Intel Xeon Silver 4210R CPU and 32 GiB of memory. Each version of the kernel uses the same compilation configuration. Specifically,

CONFIG_BPF_SYSCALL and CONFIG_BPF_JIT were enabled for the eBPF subsystem. The JIT is required for many important features in eBPF programs, e.g., calling kernel functions. We enabled CONFIG_KASAN for memory sanitation and CONFIG_KCOV to collect the code coverage. We extended Syzkaller with our design and applied our patches to the corresponding version of Linux for memory sanitation in the eBPF program. All the experiments were configured with the same parameters in QEMU configurations and base system call descriptions. Specifically, we started all experiments simultaneously and distributed the resources evenly, including 8 cores and 8 GiB of memory for each virtual machine. To reduce statistical errors, each experiment was repeated three times and executed over a period of 48 hours, and the average results were reported.

### 6.1 Bug Finding

To answer **RQ1** and evaluate the effectiveness of BVF in finding correctness bugs, we deploy Syzkaller, Buzzer, and BVF to conduct testing on the Linux upstream and bpf-next repository for two weeks. bpf-next contains the latest codebase for the eBPF subsystem and is actively developed by the maintainers. We chose those kernel versions because testing the upstream kernel is currently the best practice for bug finding, and Linux maintainers also encourage the community to conduct tests on upstream. The reasons are: 1) bugs uncovered in those latest versions are likely previously unknown and should be fixed immediately; 2) one can detect bugs that may impact various past stable versions by testing upstream; 3) testing upstream prevents bugs from impacting future releases. Furthermore, only bugs with stable reproducers that received explicit confirmations from maintainers are reported and listed. We determine a correctness bug after confirming that it is triggered either by indicator #1 or #2 and captured by our sanitation or kernel mechanisms.

As a result, Syzkaller and Buzzer did not trigger any valid correctness bugs within the two weeks, while BVF found six previously unknown correctness bugs in the verifier. In reality, the verifier has received an extensive amount of scrutiny over the years as a security-sensitive kernel component, and therefore, finding bugs in it, especially correctness bugs, is challenging for existing tools, which is the major reason for the results of Syzkaller and Buzzer. On the other hand, despite those manual efforts and numerous test cases constructed by eBPF maintainers, BVF is still capable of uncovering six new correctness bugs, demonstrating the effectiveness of the proposed indicators. Specifically, three correctness bugs (#1-3) can lead to a load of invalid eBPF programs that contain memory bugs in the kernel, BVF can detect them by leveraging the memory sanitation proposed for indicator#1. The remaining correctness bugs (#4-6) can lead to kernel deadlock or direct kernel panic after the corresponding programs are loaded, BVF can trigger those bugs efficiently because of the improved quality of the generated programs utilizing the proposed program structure, thus triggering indicator#2 effectively. Furthermore, the correctness bugs found have a wide impact, e.g., Bug#4 has existed for 4 years, and Bug#6 was fixed in upstream, and the corresponding patches were backported to Linux v6.1, v5.15, v5.10, etc. All the vulnerabilities have been confirmed and nine of them have been fixed by the maintainers and us.

**Table 2.** BVF found 11 new vulnerabilities in total, six of them are correctness bugs in the verifier. The first column shows the component in the eBPF subsystem that contains the bug and the rest of the columns illustrate the root cause and the status of the vulnerabilities respectively.

| # | Component | Description | Status |
|---|---|---|---|
| 1 | Verifier | *Incorrect nullness propagation of pointer comparisons causes invalid memory access* | Fixed |
| 2 | Verifier | *Incorrect task struct access validation leads to out-of-bound access* | Confirmed |
| 3 | Verifier | *Incorrect check on kfunc call operations causes verifier backtracking bug* | Fixed |
| 4 | Verifier | *Missing check on programs attached to bpf_trace_printk causes deadlock* | Fixed |
| 5 | Verifier | *Missing validation on contention_begin causes inconsistent lock state error* | Fixed |
| 6 | Verifier | *Missing strict checking on signal sending of programs causes kernel panic* | Fixed |
| 7 | Dispatcher | *Missing sync between dispatcher update and execution leads to null-ptr-deref* | Fixed |
| 8 | Syscall | *Incorrect using of kmemdup() leads to failure in duplicating xlated insns* | Fixed |
| 9 | Map | *Incorrect bucket iterating in the failure case of lock acquiring causes oob access* | Fixed |
| 10 | Helper | *Incorrect using of irq_work_queue in a helper function leads to lock bug* | Fixed |
| 11 | XDP | *Incorrect execution env, attempt to run device eBPF program on the host* | Confirmed |

BVF also found five vulnerabilities (#7-11) in related components in the eBPF subsystem. For instance, an improper use of kmemdup() in an eBPF system call that intends to duplicate rewritten instructions to user space processes would fail when the size of instructions exceeds the allocation limitation of kmalloc(). We submitted two patches to fix the bug. The first one introduces a new primitive called kvmemdup() that utilizes kvmalloc() to allocate memory, i.e., switch to vmalloc() in the failure case of kmalloc(), and the patch has been accepted by the mm maintainers; the second patch modifies the corresponding system calls in eBPF to utilize the new primitive, thus effectively addressing the bug. BVF is capable of detecting those five vulnerabilities because the programs generated are complex, thereby facilitating the testing of related operations on them.

### 6.2 Case Study

**Nullness Propagation (Bug#1).** After the change from commit befae75856ab of Linux, the verifier propagates nullness information while analyzing jump instructions as demonstrated in Listing 3. Specifically, for jump operations with equality comparison whose operands are both pointers, the

verifier marks the nullable pointer as non-null if it is aware that the other is not null in the corresponding equal path. For instance, for `if r0 == r1 goto +1` instruction where r0 is nullable while r1 is non-null, the verifier would mark r0 as non-null in the equal path.

```
--- a/kernel/bpf/verifier.c
+++ b/kernel/bpf/verifier.c
@@ -11822,10 +11822,17 @@ static int check_cond_jmp_op(struct
↪  bpf_verifier_env *env,
      *      register B - not null
      * for JNE A, B, ... - A is not null in the false branch;
      * for JEQ A, B, ... - A is not null in the true branch.
+     *
+     * Since PTR_TO_BTF_ID points to a kernel struct that does
+     * not need to be null checked by the BPF program, i.e.,
+     * could be null even without PTR_MAYBE_NULL marking, so
+     * only propagate nullness when neither reg is that type.
      */
     if (!is_jmp32 && BPF_SRC(insn->code) == BPF_X &&
         __is_pointer_value(false, src_reg) &&
         __is_pointer_value(false, dst_reg) &&
-        type_may_be_null(src_reg->type) !=
-        type_may_be_null(dst_reg->type)) {
+        type_may_be_null(src_reg->type)!=
+        type_may_be_null(dst_reg->type) &&
+        base_type(src_reg->type) != PTR_TO_BTF_ID &&
+        base_type(dst_reg->type) != PTR_TO_BTF_ID) {
         eq_branch_regs = NULL;
         switch (opcode) {
         case BPF_JEQ:
```

**Listing 3.** This patch fixes the correctness bug #1 directly by filtering the jump instructions with `PTR_TO_BTF_ID` pointer type. In this way, the verifier would propagate nullness information for register-to-register comparisons in jump instructions only if neither register is that pointer type.

However, the filter condition in the verifier is incomplete, leading to an error that incorrectly allows programs with invalid memory access to be loaded into the kernel. Specifically, a special kind of pointer in eBPF programs named `PTR_TO_BTF_ID` points to kernel structures that the programs do not need to conduct a null check before using, i.e., the verifier does not mark such pointer as `maybe_null` even though the pointer could be, and the null dereference of such pointers is properly handled by the kernel [10]. In the case that one pointer is `maybe_null` and the other is a pointer to BTF, the former is incorrectly marked as non-null if the latter equals zero at runtime. Listing 2 demonstrates the program generated by BVF that triggers an invalid memory access due to the mentioned flaw. After #1, the verifier tracks r6 as `PTR_TO_BTF_ID` without `maybe_null` marking, which, however, actually equals null at runtime. After #5, r0 is marked as a pointer to a map value that may be null by the verifier, meaning that it cannot be dereferenced without a null check. After #6, however, r0 is incorrectly marked as non-null due to the aforementioned nullness propagation. Both r0 and r1 are null pointers at runtime, leading to invalid memory access at #7.

We proposed two patches for this bug. The first one shown in Listing 3 fixes the bug by filtering the corresponding pointer type, and the second patch presents one test case that illustrates the correct behavior. BVF can uncover this vulnerability because the generated program passed the verifier but triggered our instrumented runtime checks, which demonstrates the effectiveness of the indicators.

### 6.3 Program Structure Effectiveness

To address **RQ2** and evaluate whether the proposed program structure can assist BVF in improving the quality of the generated eBPF programs, we deploy Syzkaller, Buzzer, and BVF, and compare their testing performance. Three versions of Linux are adopted, including 5.15, 6.1, and `bpf-next` branch. We choose `bpf-next` because it is upstream of the eBPF subsystem, and Linux v6.1 and Linux v5.15 are the two representing release versions used by many distributions. All the tools invoke the same set of eBPF system calls and only the eBPF source code is instrumented during compilation by `kcov` [8] for coverage collection so that the testing range of all the tools is the same. Finally, each testing campaign is repeated three times, and we report the final average value over a period of 48 hours.

Figure 6 illustrates the comparison of branch coverage among Syzkaller, Buzzer, and BVF. As shown in the figure, BVF can achieve the highest coverage compared to Syzkaller and Buzzer in the same amount of time. Specifically, all tools show significant growth in the first eight hours, where the improvement of BVF is not obvious. After testing for eight hours, the coverage growth of Syzkaller and Buzzer starts to slow down and tends to saturation, while BVF is significantly faster than them. This is because the proposed mechanism does not promote the throughput directly, but rather enables the fuzzer to reach more code by generating complicated eBPF programs. Therefore, all tools perform a similar rate before eight hours since they have yet to cover the code reachable using the previous mechanism. They diverge after eight hours as the random instruction selection adopted by Syzkaller and Buzzer cannot provide more low-hanging fruit, while BVF is capable of covering more code in the verifier continuously by generating interesting programs. Table 3 lists the detailed statistics of the covered branches achieved by Syzkaller, Buzzer, and BVF.

**Table 3.** Coverage statistics of Syzkaller, Buzzer, and BVF on Linux over 48 hours. The improvements of BVF over Syzkaller and Buzzer are shown in the parenthesis.

| Version | **BVF** | Syzkaller | Buzzer |
|---|---|---|---|
| v5.15 | **50192** | 41433 (+17.5%) | 9176 (+447.0%) |
| v6.1 | **67348** | 56458 (+16.2%) | 10059 (+569.5%) |
| bpf-next | **65176** | 52295 (+19.8%) | 9271 (+603.0%) |
| Overall | **60905** | 50062 (**+17.8%**) | 9502 (**+541.0%**) |

Compared to Syzkaller, BVF is capable of covering 17.8% more of the verifier's code. To reason about the result, we

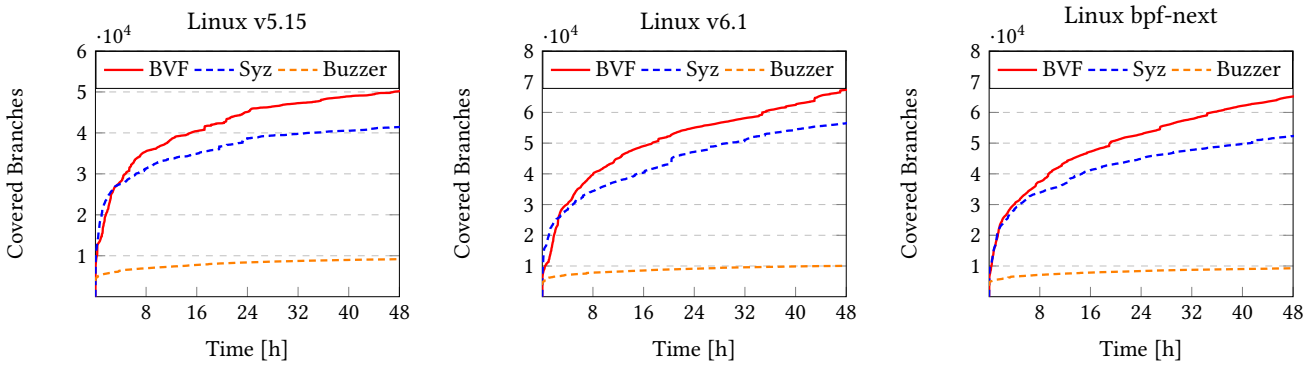


**Figure 6.** Branch coverage on Linux 5.15, 6.1, and `bpf-next` branch of Syzkaller, Buzzer, and BVF over 48 hours on average of 3 repetitions. In all three kernel versions, BVF achieves the highest coverage statistics, demonstrating that the program structure can assist BVF to synthesize effective programs to cover the verifier's code.

further evaluated the acceptance rate, i.e., comparing the programs passing the verifier with all the generated programs. As a result, the acceptance rate of BVF is more than twice higher than that of Syzkaller, where Syzkaller archives 23.5% while BVF reaches 49%. We also collected and inspected the reasons for the rejection of Syzkaller-generated programs by collecting the error code returned. The reasons are various, and the two most returned errno are EACCES and EINVAL. Therefore, the key factor of Syzkaller's lower acceptance rate is its random instruction generation, which generates many invalid instructions that perform illegal registers or memory access. In essence, the difference between BVF and Syzkaller is whether the tool utilizes the proposed structure to generate programs or not. Therefore, the results indicate that the program structure can facilitate more effective program generation, thereby achieving a higher acceptance rate and exploring more verification logic.

In comparison with Buzzer, BVF achieved 5.41× improvement in the verifier's code coverage. We also evaluated the acceptance rate of Buzzer. Specifically, since Buzzer contains two testing modes, we collected the acceptance rates accordingly and calculated their respective values, specifically 1% and 97%. However, these values should not be interpreted directly for the following reasons. For the former mode, Buzzer generates highly random programs to an extent such that very few programs pass the verifier, thus greatly hindering testing effectiveness. For the latter mode, Buzzer mainly involves ALU and JMP instructions, thus the generated programs are relatively simple (88.4%+ instructions are ALU and JMP), consequently failing to trigger bugs requiring more sophisticated logic. Unlike Buzzer, both Syzkaller and BVF consider all kinds of instructions. For instance, the programs generated by BVF may access all the accessible objects, including different map types and kernel objects, with various load/store instructions, and invoke the available eBPF helper functions. In addition, BVF achieves a 49% acceptance rate with these capabilities, therefore, the generated programs of BVF are much more expressive while comparably successful in acceptance, resulting in coverage improvement.

Therefore, BVF's program structure allows for generating interesting programs, thereby achieving higher coverage and triggering more correctness bugs.

### 6.4 Sanitation Overhead

To address **RQ3** and evaluate the overhead of our memory sanitation, we measure the execution time and the number of instructions before and after instrumentation. The datasets used in this experiment contain the manual-written eBPF programs in the verifier's self-tests. Those programs are representative given that they are carefully encoded by eBPF maintainers to cover a wide range of scenarios. Also, tests without any load/store are skipped since they cannot trigger our instrumentation, and the final number of eBPF programs for evaluation is 708. Finally, the measurements are performed three times and the average results are reported.

As a result, the average slowdown in execution speed is 90% and the increase in instruction footprints is 3.0x caused by the instrumentation. This is mainly due to the additional instructions instrumented for sanitation. We compare our approach to ASAN [34], as eBPF programs are fairly self-contained, and thus more similar to user programs than a kernel. Based on the extensive evaluation of ASAN, the average slowdown caused by its instrumentation on CPU2006 benchmarks is 73% and the average memory consumption caused by the instruction footprints is 3.37x. The average slowdown introduced by BVF's sanitation is relatively higher than that of ASAN. This is mainly because BVF relies on KASAN runtime to collect memory states and perform sanitation, which is more complex than ASAN runtime designed for user space programs. The amount of instrumented instructions caused by BVF's sanitation is lower than that of ASAN because the instruction set designed for eBPF programs is much simpler than that of programs compiled to different hardware architectures. We conclude that the overhead introduced by BVF's instrumentation is well within expectations and reasonable,

given its ability to capture bugs in eBPF programs and assist in correctness bug finding.

### 6.5 Discussion

**False Positives/Negatives.** As mentioned in Section 5, we implement memory sanitation by dispatching the load/store to the kernel functions instrumented by KASAN. Since all the bugs are detected dynamically, BVF experiences a low probability of false positives and we didn't find such cases during the experiment. On the other hand, the possible existence of false negatives in KASAN may be propagated to the testing campaign of BVF, potentially leading to missed correctness bugs in the verifier. At present, we are unable to address this issue with effective solutions within BVF itself, and it requires future improvements to KASAN.

**Detectable Bug Types.** As is the general situation in dynamic testing, we do not claim completeness, but BVF is capable of capturing various exceptional behaviors caused by erroneous instructions generated through incorrect analysis for the following reasons: 1) Correctness bugs in load and store analysis can lead to programs with memory bugs being loaded, thus captured by BVF's sanitation, e.g., Bug#2; 2) Incorrect analysis of ALU operation can be propagated to load/store or jump, thus being detected, e.g., Bug#6 and CVE-2022-23222; 3) Bugs of jump analysis mainly consist of incorrect branch analysis and insufficient function call checking, where the former leads to illegal register states and thus can be propagated to load/store and captured, e.g., Bug#1, and the latter can be captured during execution in kernel routines, e.g., BUG#4.

**Bug Triage.** Although BVF can detect correctness bugs automatically, we still rely on manual methods for locating and analyzing the verifier's bugs. Specifically, the amount of effort required to identify the verifier bug given an eBPF program that has an error but passes the verifier is as follows. In practice, we triage the bugs triggered, manually inspect the erroneous eBPF programs to pinpoint the guilty instruction, and then reason the preceding instructions to collect related operations that produce the operands for the guilty instruction. The guilty instruction and related operations together enable us to locate the possible incorrect verifying logic according to their instruction kinds. Finally, maintainers and we will look into those parts of the verifier and analyze the root cause.

**Reachable Code.** In essence, the program generation of BVF can cover a wide range of possible programs because: 1) all possible instruction types are supported; 2) all possible combinations of instructions can be constructed in different parts of the framed body. The second holds because while BVF encodes certain operation patterns, it also supports stochastic instruction selection. Meanwhile, we also intentionally choose not to generate certain kinds of programs, e.g., programs using uninitialized registers or consisting of outbound jump, because these programs can be rejected easily. Therefore, BVF is capable of covering various functionalities of the verifier. The functionalities not covered mainly consist of 1) basic instruction validity checks since we avoid generating such programs; and 2) verifier logging code since we currently do not parse the log.

**Future Applicability.** While maintainers modify eBPF with new verifier algorithms and data structures, the idea of converting the correctness bug detection into the bug-finding in the eBPF program is generalizable as the two major sources for correctness bugs (indicators#1 and #2) are already captured. Therefore, correctness bugs included in the newly added code will eventually appear as the two indicators, thus captured by the corresponding mechanisms. For API changes, we can support them once the relevant system call descriptions (actively maintained by the kernel community) are added.

## 7 Related Work

Fuzz testing is an effective bug-finding approach, and many works in this area are relevant to our work. Specifically, Syzkaller [43], the state-of-the-art kernel fuzzer, is capable of testing the eBPF subsystem by generating random `bpf()` system calls. It has been integrated into eBPF upstream testing [42] and has demonstrated promising capability in uncovering memory errors. However, the inputs generated by syzbot are likely to have invalid bytes, and valid instructions in them can violate simple rules of eBPF programs, e.g., using registers without initialization thus would be early rejected by the verifier. Furthermore, even if an eBPF program that passes the verifier's checks is generated, detecting correctness bugs is challenging for syzbot or similar testing tools. Because, unlike detecting memory errors, where the sanitizers can signal the fuzzer once capturing such bugs, correctness bugs typically do not result in direct kernel crashes or observable outputs, and therefore can be omitted due to the absence of checking mechanisms.

Another relevant work [23] proposed by iovisor is dedicated to testing the verifier. It first ports the verifier from kernel space to user space by substituting related kernel routines with simplified versions, and then, tests the verifier with libfuzzer [32] by generating random byte sequences. As a result, it reported one memory bug in the verifier. However, the porting approach is hardly compatible with eBPF development, e.g., each change in dependent kernel routines requires modifications. In addition, it aims to detect memory errors, not verifier's correctness bugs. Buzzer [24] is a recent work that also targets the verifier. It randomly constructs eBPF programs containing certain map operations and tests the verifier by checking relevant map states. However, its generation algorithm mainly involves simple ALU and JMP instructions, leaving much of the sophisticated checking logic in the verifier less tested. As a result, Buzzer only found

one bug in the verifier's backtracking procedures by the community after a prolonged time of testing.

Unlike the aforementioned works, we propose BVF to detect the verifier's correctness bugs rather than traditional bug finding in the eBPF subsystem. To achieve this, an effective test oracle is proposed. Specifically, we first propose a lightweight structure to guide the generation of eBPF programs, thereby bypassing the verifier efficiently. We then devise two indicators for correctness bugs based on eBPF programs' intrinsic behaviors and capture the indicators with the corresponding mechanisms.

In addition to the aforementioned works, we further classify related works into the following categories.

**Formal Verification.** Formal verification checks each possible execution of a program against the specification, thus providing the strongest guarantee of correctness. Recent works apply such a technique in different components of the eBPF subsystem. Specifically, Agni [41] generates formulas representing the verifier's range analysis from source code and utilizes SMT solvers for correctness checking. Jitterbug [30] encodes specifications and semantics to check the correctness of the eBPF JIT compiler. Both advance the correctness of the eBPf subsystem significantly. Nevertheless, despite the strong guarantees of formal verification, dynamic testing is able to perform continuous testing and discover numerous bugs along with the program evolving and changing with less manual effort, thus being practical and scalable. Compared to the works mentioned, BVF is capable of continuously detecting various correctness bugs in the entire verifier, not just range analysis (a small portion of the verifier), without manual intervention.

**Black Box Testing.** Works in this category regard the system under test as a black box, and conduct testing by continuously generating inputs of interest. Take CSmith [47] for instance. It generates a subset of C programs that is free of undefined behaviors such that the generated programs can be utilized for differential testing to detect compiler bugs. In comparison, BVF designs structured generation based on intrinsic behaviors of eBPF programs, synthesizes programs that contain various map/context/memory accesses and active interactions with the kernel, and detects correctness bugs using effective oracles without requiring any referencing implementations. Both the generating and bug-finding approaches are different.

**Sanitizer Design.** Recent advances in sanitizer design greatly facilitate bug finding for dynamic testing. Such a technique is capable of automatically detecting certain kinds of bugs by instrumenting the program under test. For instance, Asan [34] captures memory bugs by collecting memory states and checking sensitive operations in a program with the instrumented code. However, sanitizers only passively detect bugs. In BVF, we convert the correctness bug finding into eBPF program bug detection by utilizing existing sanitizer infrastructure to capture anomalous behaviors from erroneous programs proactively generated by BVF.

## 8 Conclusion

In this paper, we propose two indicators and corresponding capturing mechanisms as effective test oracles for the verifier's correctness bugs. To trigger the indicators, BVF leverages the proposed structure to synthesize complex programs. To capture the indicators, BVF executes the programs and utilizes the memory sanitation and kernel mechanisms. As a result, BVF detected 11 previously unknown vulnerabilities in the eBPF subsystem, and six of them are correctness bugs in the verifier, demonstrating that our technique is highly effective in finding correctness bugs.

## Acknowledgments

We would like to thank our shepherd, Pedro Fonseca, and the anonymous Eurosys reviewers for valuable feedback and input on this paper. This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000) and NSFC Program (No. 62022046, 92167101, U1911401, 62021002).

## References

[1] CVE-2021-3490. https://nvd.nist.gov/vuln/detail/CVE-2021-3490.
[2] CVE-2021-4159. https://nvd.nist.gov/vuln/detail/CVE-2021-4159.
[3] CVE-2022-23222. https://nvd.nist.gov/vuln/detail/CVE-2022-23222.
[4] CVE-2022-23222 exploit. https://www.openwall.com/lists/oss-security/2022/01/18/2.
[5] eBPF Instruction Set Specification. https://docs.kernel.org/bpf/instruction-set.html.
[6] ebpf selftests. https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf-next.git/tree/tools/testing/selftests/bpf.
[7] eBPF Verifier. https://docs.kernel.org/bpf/verifier.html.
[8] kcov: code coverage for fuzzing. https://docs.kernel.org/dev-tools/kcov.html.
[9] Oss-fuzz: continuous fuzzing for open source software. https://github.com/google/oss-fuzz.
[10] PTR_TO_BTF_ID. https://github.com/torvalds/linux/blob/v6.2/include/linux/bpf.h#L760.
[11] Runtime locking correctness validator. https://docs.kernel.org/locking/lockdep-design.html.
[12] Andrea Arcangeli. Seccomp bpf (secure computing with filters). https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html?highlight=seccomp.
[13] Marco Bonola, Giacomo Belocchi, Angelo Tulumello, Marco Spaziani Brunella, Giuseppe Siracusano, Giuseppe Bianchi, and Roberto Bifulco. Faster software packet processing on FPGA NICs with eBPF program warping. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 987–1004, Carlsbad, CA, July 2022. USENIX Association.
[14] Daniel Borkmann and Alexei Starovoitov. Linux eBPF. https://ebpf.io.
[15] Peng Chen and Hao Chen. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725, 2018.
[16] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers. In *28th USENIX Security*

*Symposium (USENIX Security 19)*, pages 1967–1983, Santa Clara, CA, August 2019. USENIX Association.

[17] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and precise static analysis of untrusted linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 1069–1084, New York, NY, USA, 2019. Association for Computing Machinery.

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

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

[20] Tejun Heo. sched: Implement BPF extensible scheduler class. https://lwn.net/Articles/916290/.

[21] W.E. Howden. Theoretical and empirical studies of program testing. *IEEE Transactions on Software Engineering*, SE-4(4):293–298, 1978.

[22] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. Difuzzrtl: Differential fuzz testing to find cpu bugs. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1286–1303, 2021.

[23] iovisor. bpf-fuzzer: fuzzing framework based on libfuzzer and clang sanitizer. https://github.com/iovisor/bpf-fuzzer.

[24] Juan José López Jaimez and Meador Inge. Buzzer. https://github.com/google/buzzer.

[25] Jim Keniston. Linux Kprobe. https://www.kernel.org/doc/html/latest/trace/kprobes.html.

[26] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 147–161, New York, NY, USA, 2019. Association for Computing Machinery.

[27] Hsuan-Chi Kuo, Kai-Hsun Chen, Yicheng Lu, Dan Williams, Sibin Mohan, and Tianyin Xu. Verified programs can party: Optimizing kernel extensions via post-verification merging. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 283–299, New York, NY, USA, 2022. Association for Computing Machinery.

[28] lcamtuf. American fuzzy lop, 2013. https://lcamtuf.coredump.cx/afl/.

[29] J. Liang, M. Wang, C. Zhou, Z. Wu, Y. Jiang, J. Liu, Z. Liu, and J. Sun. PATA: Fuzzing with Path Aware Taint Analysis. In *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*, pages 154–170, Los Alamitos, CA, USA, may 2022. IEEE Computer Society.

[30] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. Specification and verification in the field: Applying formal methods to bpf just-in-time compilers in the linux kernel. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[31] 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.

[32] LLVM Project. libfuzzer: a library for coverageguided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[33] 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.

[34] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 28, USA, 2012. USENIX Association.

[35] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, page 62–71, New York, NY, USA, 2009. Association for Computing Machinery.

[36] Evgeniy Stepanov and Konstantin Serebryany. Memorysanitizer: fast detector of uninitialized memory use in c++. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 46–55. IEEE, 2015.

[37] 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, Carlsbad, CA, July 2022. USENIX Association.

[38] 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.

[39] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Comput. Surv.*, 53(1), feb 2020.

[40] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. Sound, precise, and fast abstract interpretation with tristate numbers, 2021.

[41] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. Verifying the verifier: ebpf range analysis verification. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification*, pages 226–251, Cham, 2023. Springer Nature Switzerland.

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

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

[44] Dmitry Vyukov and Andrey Konovalov. Syzlang: System Call Description Language, 2015. https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions_syntax.md.

[45] 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.

[46] 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, 2020.

[47] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 283–294, New York, NY, USA, 2011. Association for Computing Machinery.

[48] Masanobu Yuhara, Brian N. Bershad, Chris Maeda, and J. Eliot B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *USENIX Winter 1994 Technical Conference (USENIX Winter 1994 Technical Conference)*, San Francisco, CA, January 1994. USENIX Association.

[49] Jiang Zhang, Shuai Wang, Manuel Rigger, Pinjia He, and Zhendong Su. SANRAZOR: reducing redundant sanitizer checks in C/C++ programs. In Angela Demke Brown and Jay R. Lorch, editors, *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pages 479–494. USENIX Association, 2021.

[50] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. XRP: In-Kernel storage functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 375–393, Carlsbad, CA, July 2022. USENIX Association.

[51] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: A survey for roadmap. *ACM Comput. Surv.*, 54(11s), sep 2022.