Toss a Fault to BpfChecker: Revealing Implementation Flaws for eBPF runtimes with Differential Fuzzing

Chaoyuan Peng*
Zhejiang University
Hangzhou, China
ret2happy@zju.edu.cn

Muhui Jiang
The Hong Kong Polytechnic University
Hong Kong, China
muhjiang@polyu.edu.hk

Lei Wu
Zhejiang University
Hangzhou, China
lei_wu@zju.edu.cn

Yajin Zhou†
Zhejiang University
Hangzhou, China
yajin_zhou@zju.edu.cn

ABSTRACT

eBPF is a revolutionary technology that can run sandboxed programs in a privileged context and has an extensive range of applications, such as network monitoring on Linux kernel, denial-of-service protection on Windows, and the execution mechanism of smart contracts on blockchain. However, implementation flaws in eBPF have broad-reaching impact and serious consequences. Prior studies primarily focus on the memory safety of the eBPF runtimes, but few can detect implementation flaws (i.e., whether the implementation is correct). Meanwhile, existing implementation flaws detecting methods predominantly address bugs in the verifier, neglecting bugs in other components (i.e., the interpreter and the JIT compiler). In this paper, we present BpfChecker, a differential fuzzing framework to detect implementation flaws in the eBPF runtimes. It utilizes eBPF programs as input, performing differential testing for the critical states across various eBPF runtimes to uncover implementation flaws. To enhance the semantics of generated programs, we devise a lightweight intermediate representation and perform constrained mutations under the guidance of error messages. We have implemented a prototype of BpfChecker and extensively evaluated it on the three eBPF runtimes (i.e., Solana rBPF, vanilla rBPF, Windows eBPF). As a result, we have uncovered 28 new implementation flaws, received 2 CVEs and $800,000 bounty with developers' acknowledgment. More importantly, 2 of the newly found bugs can be used to create divergences in the execution layer of the Solana network.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

*Part of the work was accomplished when the author is a research intern at BlockSec.
†Corresponding Author

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

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690237

KEYWORDS

Differential Fuzzing, eBPF, Software Security

ACM Reference Format:
Chaoyuan Peng, Muhui Jiang, Lei Wu, and Yajin Zhou. 2024. Toss a Fault to BpfChecker: Revealing Implementation Flaws for eBPF runtimes with Differential 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/10.1145/3658644.3690237

1 INTRODUCTION

eBPF (extended Berkeley Packet Filter) [4] is a promising technology originating in the Linux kernel that allows (extensible) programs to run in a privileged context, like an operating system kernel. Leveraging its programmability, eBPF is widely adopted for different applications in industry and academia [53]. Cloud service providers, for instance, employ eBPF for efficient packet filtering [1, 17, 46] and load balancing [14, 50]. Windows operating system uses eBPF to provide observability and denial-of-service protection [5, 18]. In addition, Solana [49], a rapidly growing high-performance blockchain, utilizes a variant of eBPF as the execution environment for on-chain programs (or smart contracts [2]). Given the widespread adoption and thriving ecosystem of eBPF, any bugs in the *eBPF runtime* would have a broad-reaching impact and serious consequences.

Unfortunately, this concern is valid: real-world bugs in the eBPF runtimes pose significant risks. The implementation of the eBPF runtime may not conform to the eBPF specifications, and these *implementation flaws* can produce unexpected execution results. For instance, implementation flaws [21, 23, 25] within the kernel eBPF can lead to privilege escalation issues. In the Solana blockchain, different execution results between the JIT compilers and interpreters can produce inconsistent results among validators in the network, leading to the network partition in the worst case (e.g., CVE-2021-46102 [24] and CVE-2022-23066 [22]). Hence, comprehensively detecting implementation flaws in all components in eBPF runtimes is significant since these undetected (*silent*) bugs will provide essential exploitation primitives.

Various fuzzing tools [15, 28, 30] and verification techniques [26] have emerged and aim to uncover bugs in the eBPF framework. In particular, coverage-guided fuzzers like Syzkaller [7] and BRF [11] generate system-call-based test cases and have identified memory

bugs in the Linux kernel eBPF. These approaches find it challenging to detect logic bugs as they use crashes or the sanitizer violation as the sole oracle, namely, rules to determine whether the execution of a program is correct. Though some previous works have applied semantic rules as their oracle, they are exclusively designed for verifiers. For instance, Scannell proposed an eBPF fuzzer [35], which generates template programs that write values to the calculated map pointer and detect faulty pointer arithmetic bugs of the verifier by checking the value of the map. Other semantic-aware testing tools [9, 41] check the correctness of the verifier assumptions by observing the execution state of the sanitized eBPF program. The formal verification technique [27] is also proposed to verify the eBPF JIT compiler by the stepwise specification. It requires reference implementation as precise as the targeted system and also requires further efforts to extend to the other eBPF runtimes.

Current Challenges These approaches have proven effective in detecting specific types of vulnerabilities in their targeted component of the eBPF framework. However, they are insufficient for detecting implementation flaws in the user-space eBPF runtime due to several *key challenges to detecting implementation flaws for eBPF runtimes*, which existing solutions fail to address.

First, there is little prior knowledge on how to design the semantic-aware oracle to check the correctness of the entire eBPF runtimes *(Challenge 1)*. Specifically, existing eBPF fuzzers can detect memory bugs with the assistance of existing memory sanitizers [36, 42], which instrument memory-related instructions and check whether the access is deemed legitimate at runtime. These memory sanitizers are feasible mainly because deciding the ground truth of the valid memory region and the property to check is relatively trivial. However, detecting implementation flaws in the eBPF runtime requires that the oracles contain fine-grained state model with eBPF domain knowledge. Due to the subtle design disparities across different runtimes, the modeled oracles must avoid semantics conflicts. However, to the best knowledge, no existing work models and aligns states of different eBPF runtimes. **Second**, it is generally challenging for dynamic techniques to generate representative eBPF programs that encompass both legal and illegal ones to explore *all eBPF components*, namely parser, verifier, and executor *(Challenge 2)*. Since the eBPF runtime only executes the verified eBPF program, existing template-based and grammar-aware techniques generate programs that are likely to be rejected by the eBPF runtime (as they can't pass the verification) and thus are not representative. **Third**, new indicators for the eBPF runtime are required for measuring runtime states and providing effective feedback *(Challenge 3)*. Code coverage does not inherently indicate the efficacy of a given eBPF program when a subtle bug is triggered by a specific value or control flow [29, 48]. Existing eBPF fuzzing techniques (e.g., [7, 9, 11]) leverage coverage-driven mutation to generate new test cases. However, when two eBPF programs cover different internal states of the runtime, they can still reach similar code coverage, making the coverage metric not sensitive to the concrete value of the instruction [10]. Consequently, we need a new feedback metric to indicate whether the generated test case is effective.

Our solutions To address the above challenges, we propose BpfChecker, the first general differential fuzzing framework for detecting implementation flaws in eBPF runtimes. The main insight behind BpfChecker is that the inconsistent intermediate states across different eBPF runtimes generally indicate potential implementation flaws. In order to address *Challenge 1*, BpfChecker models the intermediate states and compares the post-states across different user-space eBPF runtimes after program execution to detect implementation flaws. To get a deterministic execution result and eliminate false positives, we provide the same context before executing a given eBPF program on different eBPF runtimes. We use R to denote a specific eBPF runtime and P to denote the input eBPF program. The captured intermediate state after the program execution can be represented as $S_E = R(P)$. We identified the potential implementation flaw when the checking procedure for n runtimes is met:

$$\forall i, j \in \{0, 1, \ldots, n\}, \quad \text{if } R_i(P) \neq R_j(P)$$

The intermediate states S_E include the error message from the verifier and the concrete step-wise execution states from the executor (i.e., accessible memory content and the values of registers).

In order to address *Challenge 2*, we propose a lightweight intermediate representation (IR) with eBPF-affinity and generate the semantically constrained eBPF program with different styles. The state-of-the-art methods (e.g., [7, 9]) leverage grammar-aware or template-based program generation. Therefore, they achieve a low semantic correctness rate of the generated program. Existing LLVM IR does not support variants of the eBPF instruction set for different eBPF runtimes and may produce redundant instructions. Moreover, it is challenging to leverage LLVM IR to generate illegal instructions. While enhancing the LLVM's eBPF backend is feasible, it requires substantial effort. Our proposed IR not only supports various eBPF instruction sets but flexibly generates low-level instructions as well. Specifically, each IR instruction includes an operator, an offset value, an immediate value, and no more than two operands. In order to create a sophisticated program control flow, the basic blocks of the IR program are designed to be connection-free. The evaluation shown in 4.1 demonstrates that with the help of our lightweight IR, the generated eBPF program can reach an average 18.28% semantic correctness rate, which is 2.1× of the baselines.

In order to address *Challenge 3*, we utilize the error message during the execution in the user-space eBPF runtimes as feedback and perform IR program correction upon the generated program. Suppose the error message from the verifier shows that the invalid memory access is due to the second instruction. In that case, mutating the second instruction is more likely to turn this program into a valid one than mutating other instructions. We implement different mutators to handle the common verifier error messages (e.g., the memory access violation error, PC error, etc.). The evaluation shown in 4.1 demonstrates that the semantic IR corrector could improve the semantic correctness rate of the generated program by around 5%. It is noteworthy that BRF conducts error message-driven mutations at the source code level, whereas BpfChecker engages in low-level instruction mutations, which are considerably more complex and effective than those at the source code level.

We implement our prototype of BpfChecker and evaluate it on three eBPF runtimes. The results show that BpfChecker can generate sufficient test cases, which are all syntactically correct

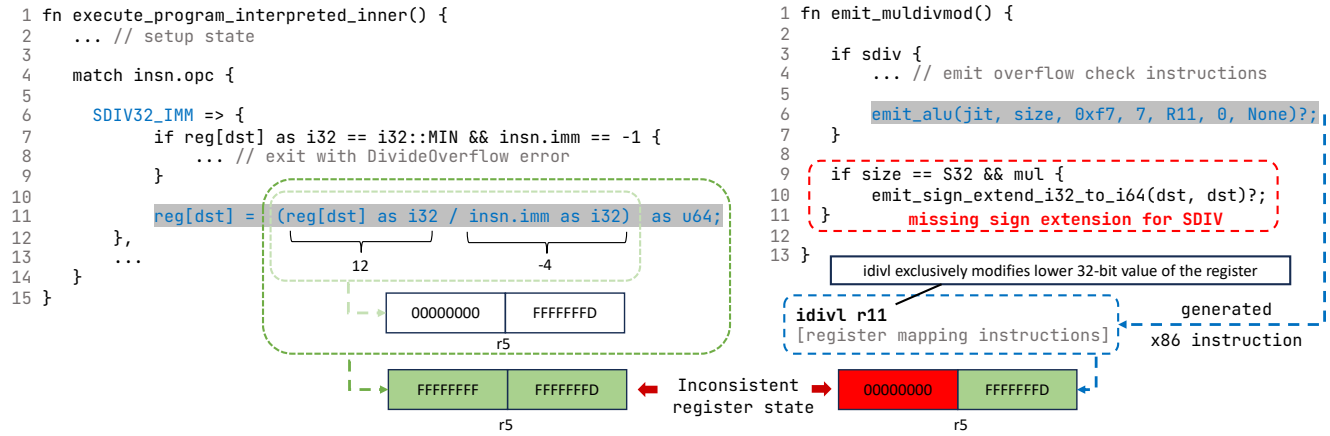

Figure 1: Motivating example. The left part shows the correct implementation of the SDIV instruction for the interpreter. The right part shows the buggy implementation during native code generation for SDIV instruction in the JIT compiler. The inconsistent register state after the execution reveals the implementation flaws in the Solana rBPF engine.

instructions and achieve 2.1× semantic correctness of the baselines. Furthermore, our proposed lightweight eBPF IR can improve the semantic correctness by at least 34%. A total of **28** bugs were found, and 18 were fixed. Moreover, we receive 2 CVEs and $800,000 bounty from the vendor. An analysis of the detected bugs indicates that 18 of the detected bugs do not manifest as segmentation faults or sanitizer violation conditions; thus, they cannot be detected using state-of-the-art methods. BpfChecker is available at https://github.com/bpfchecker/BpfCheckerSource.

In summary, we make the following contributions:

- **Sufficient test case generator** We propose a semantic-aware test case generator that leverages lightweight eBPF IR under error message guidance to generate eBPF programs with a high semantic correctness rate.
- **Deterministic differential testing engine** We model the intermediate states of the eBPF runtimes and propose the semantic oracle, which detects the implementation flaws by comparing the execution states among each runtimes.
- **Effective prototype system and new findings** We present the design and implementation of BpfChecker, a differential fuzzing framework to detect implementation flaws in the eBPF runtimes. We further apply BpfChecker against three eBPF runtimes, exploring the inconsistency and crash in each implementation. As a result, we identified 28 new bugs from three eBPF runtimes and received a total of $800,000 bounty. Two of the newly found bugs are critical severity, and two CVEs have been assigned so far. More importantly, to our best knowledge, one of the bugs is the first publicly disclosed consensus bug in the Solana execution layer.

2 BACKGROUND AND MOTIVATION

2.1 eBPF

eBPF (Extended Berkeley Packet Filter) has evolved significantly and now offers a more general-purpose framework for running custom code in the privileged space: its execution is strictly *sandboxed*, ensuring no unexpected side-effects can manifest. While the initial implementation of eBPF is deeply integrated within the Linux kernel, the demand for its generic programmability and functionality in user-space applications leads to the evolution of user-space eBPF runtimes [32, 34]. These emerging runtimes allow developers to leverage eBPF's generic features beyond the traditional kernel boundary, thereby broadening its application from innovative security measures to the smart contract execution layer.

eBPF runtime is primarily constituted by the program verifier and the executor. In order to safeguard the security and stability when executing eBPF programs, eBPF runtimes use different security approaches. User-space eBPF runtimes, such as rBPF [34] and uBPF [32], use a basic verifier to check whether the program conforms to basic syntactic specifications and enforce the legitimacy of instructions during execution. Since the executor does not enforce type or memory safety during execution, the kernel-space eBPF runtime utilizes a sophisticated verifier to check the safety of the program in terms of data flow and control flow before loading it. Once the program passes the verification, eBPF utilizes the JIT compiler to compile it to native code for execution or, alternatively, executes it via an interpreter.

eBPF verifier The eBPF verifiers utilize static analysis to check the safety of the provided programs at varying degrees of strictness. Specifically, kernel verifier [13] and *PREVAIL* verifier [44] use abstract interpretation with several abstract domains [8, 47]. These verifiers determine safety in two steps. First, they validate the control flow of a program, such as unbounded loop and unreachable instruction. Second, they simulate the execution of each instruction based on the collected states and reject the programs containing illegal operations, such as using uninitialized registers and out-of-bounds accesses.

On the other side, verifiers embedded in the user-space runtimes [32, 34] perform lightweight verification and check instruction syntactically on the operands and operators without context information. They rely on dynamic analysis to ascertain the presence of runtime security vulnerabilities within the program.

eBPF execution The typical eBPF architecture is the 64-bit register-based virtual machine with a fixed stack, while the heap space is

```
lddw r5, 0x10000000c
sdiv32 r5, -4
// Missing sign bit extension in JIT mode:
//  JIT:         r5 = 0x00000000FFFFFFFD
//  interpreter: r5 = 0xFFFFFFFFFFFFFFFD
jslt r5, 0, lbb_7
lddw r0, 0x1
lbb_7:
    exit

// Final inconsistent register states:
//  JIT:         r0 = 0
//  interpreter: r0 = 1
```

Figure 2: The simplified proof-of-concept eBPF program of CVE-2022-23066 in Solana rBPF runtime.

not contained in the specification. As an alternative, the Linux kernel provides an interface to key-value maps for persistent storage between invocations, while user-space eBPF runtime doesn't necessarily implement it. In the supported host machine architecture, Linux kernel eBPF uses the JIT compiler by default. On the other user-space eBPF runtime, utilizing the JIT compiler or the interpreter, is typically configurable. The JIT compilers in user-space eBPF runtime do not optimize the code in the same way as contemporary engines, such as V8 engine [43] or Java Virtual Machines [40]. They translate each basic instruction into native code with one pass, supplementing it with runtime error-handling code.

2.2 Motivation

In this section, we follow the discussion of Section 1 and elucidate with a real-world vulnerability of the Solana rBPF engine. We illustrate our key insight that different runtime states imply potential implementation flaws and discuss the limitations of existing work. This dangerous *zero-day* miscalculation flaw found by BpfChecker can propagate into a more observable and threatening network consensus issue in Solana. We have responsibly reported this new vulnerability to the developers and actively helped them fix it. Notably, we identified this vulnerability in a timely manner before the buggy implementation was formally switched by the Solana mainnet, thereby safeguarding the execution security of countless valuable contracts running on the Solana blockchain. Up to now, this *critical* vulnerability has been fixed by the developers and issued with a CVE ID.

As discussed in Section 1, it is non-trivial to capture this implementation flaw. Figure 1 shows the inconsistent implementation between the interpreter and JIT compiler of the Solana rBPF engine, as highlighted in red for the differences. The code spanning lines 9 to 11 on the right part of the Figure 1 manifests the bug in the JIT compiler, which doesn't generate the sign extension instruction after the `idivl` x86 instruction. In the implementation of the interpreter (i.e., line 11 on the left part of the Figure 1), it explicitly extends the 32-bit value to the 64-bit value for the destination register. This incorrect code generation leads to the inconsistent register state after execution and can be propagated into a more observable erroneous result, which is demonstrated as a proof-of-concept eBPF program.

The PoC program of the bug has been shown in Figure 2. The program initially loads a value, where the most significant 32 bits are non-zero, into register `r5`. Subsequently, it performs the `sdiv` operation and directs the execution flow based on the sign bit of register `r5`. The final register states and execution flow show a large difference under interpreter mode and JIT compilation mode. The root cause is that during the JIT compilation, it doesn't generate instructions for sign-extension of the result register. This caused the upper 32 bits of the `sdiv` result to remain zero, leading to a minor error in the calculation result. We highlight the value of `r0` and the `r5` register with code comments and append the correct calculated result by the interpreter. We present our key insight into the correctness oracle by *modeling the key state of the runtime* as the execution behavior of a program. After the program is executed in the eBPF runtimes, the key state (e.g., registers, memory) among different runtimes should be the same since they are expected to conform to the eBPF specification.

Existing approaches (e.g., BRF [11], BVF [15], buzzer [9]) can not detect this *(non-crash)* bug as they use crashes or memory safety properties as their sole oracle. Moreover, this miscalculation flaw does not violate memory-safety properties but a domain-specific correctness oracle, which is difficult to obtain in practice. First, they heavily rely on observing the memory violation, while this miscalculation is identified as the register state properties violation. Second, these techniques are mostly coverage-driven; such a test case generation strategy is unaware of whether the input is high-quality for the eBPF runtime and is unlikely to generate input that satisfies the semantic requirement of the PoC program. Additionally, static differential analysis (e.g., ParDiff [52]) is incapable of analyzing the compiled native code and exploring the infinite program states.

To tackle the aforementioned problems, critical are generating high-quality eBPF programs and designing the correctness oracle. Therefore, BpfChecker first leverages the lightweight eBPF IR to generate the test case under error message guidance. Afterward, BpfChecker models the key states of the eBPF runtimes and uses the consistences of the key states as a correctness oracle. We will demonstrate more details in the next section.

3 DESIGN

Figure 3 depicts the overall workflow of BpfChecker with the constraint-based test case generator and the differential testing engine. Firstly, we generate IR constraints that fit the requisites of two user-space eBPF verifiers (i.e., rBPF verifier and PREVIAL verifier [44]). Subsequently, the constraint-based *test case generator* (Section 3.1) generates the IR program and converts it to the eBPF program. Finally, the *differential testing engine* (Section 3.2) executes the eBPF test case, takes the error message from the rBPF verifier as the feedback, collects the intermediate states, and detects the difference between runtimes. During the fuzzing loop, the semantic IR corrector (Section 3.1.3) mutates the first instruction, resulting in the verifier rejection as exactly as possible. We find a potential bug if the differential testing engine detects disparate states. Otherwise, we save the test case to the corpus queue for further mutation if it is verified or triggers a new error message.

3.1 Constraint-based Test Case Generator

Generating programs under semantic-aware constraints requires context information of the program instructions. Hence, we devise

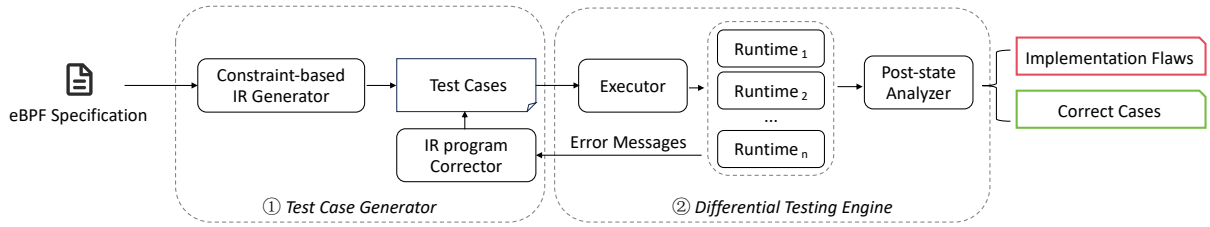

Figure 3: Overview of our design.

```
ir0<lddw,   r5, 0x10000000c>
ir1<sdiv32, r5, -4>
ir2<jslt,   ir1, 0, ir4>
ir3<lddw,   r0, 0x1>
ir4<exit>
```

Figure 4: An example IR program

an IR with eBPF-affinity (Section 3.1.1). The test case generator consists of the constraint-based generator (Section 3.1.2) and the semantic IR corrector (Section 3.1.3).

3.1.1 Intermediate Representation. Although eBPF programs can be written in C and compiled into bytecode using LLVM, the instructions generated by LLVM IR exhibit a degree of redundancy, and it is difficult to include invalid low-level instructions. Since LLVM IR is in strict SSA form, we cannot customize register usage. Moreover, different user-space eBPF runtimes use slightly varied eBPF instruction sets, which are not supported by the LLVM's eBPF backend. Enhancing the LLVM's backend to accommodate these changes requires considerable engineering effort.

To generate the program in a more low-level way and facilitate further semantic-aware program generation and correction, we propose a lightweight IR with eBPF-affinity, which ensures syntactic correctness and is tagged with the semantics of the eBPF program. It represents the eBPF program at the basic block level, which consists of detailed IR instructions. All the basic blocks are connected by each other, which forms a tree view of the program.

Syntactic Structures The eBPF instruction set constitutes a reduced instruction set architecture. The instruction fields are fixed-length. Accordingly, each IR instruction includes an operator, an offset value, an immediate value, and no more than two operands. To create a sophisticated program control flow, unreachable basic blocks are allowed to be generated. All IR instructions are contained within basic blocks, and thus, we can construct the final eBPF bytecode by traversing all these basic blocks with one pass.

Figure 4 demonstrates the lightweight IR program of the PoC program in Figure 2. Specifically, one of the operands in $ir2$ reuses the $ir1$, indicating that the result register value of the $ir1$ is used again in $ir2$. With a set of well-defined IR instructions, our IR program produces syntax-correct test cases. These test cases might contain semantic errors. Those semantic errors are introduced by the non-constrained generation part, which might use invalid values that violate the verifier rules. Next, we will fix these errors to improve the semantic correctness of the test cases.

```
BPF_MOV64_IMM(BPF_REG_2, 1)
BPF_ALU64_IMM(BPF_LSH, BPF_REG_2, 32)
BPF_ALU64_IMM(BPF_NEG, BPF_REG_2, 0)
BPF_ALU64_IMM(BPF_NEG, BPF_REG_2, 0)
BPF_JMP_IMM(BPF_JGE, BPF_REG_2, 1, 1)
BPF_RAW_INSN(BPF_JMP | BPF_EXIT, 0, 0, 0, 0)
BPF_JMP32_IMM(BPF_JLE, BPF_REG_2, 1, 1)
BPF_RAW_INSN(BPF_JMP | BPF_EXIT, 0, 0, 0, 0)
BPF_MOV32_REG(BPF_REG_2, BPF_REG_2)
// verifier: 1, truth: 0
BPF_ALU64_IMM(BPF_MUL, BPF_REG_2, -1)
// verifier: -1, truth: 0
BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, 1)
// verifier: 0, truth: 1
```

Figure 5: PoC program for CVE-2021-31440

3.1.2 Constraint-based Generator. We study diverse verification rules across the user-space eBPF verifiers (i.e., rBPF, PREVAIL verifier) and distill common constraints that preempt the generation of low-quality programs. The constraints are classified as follows.

Constraints on opcodes. We construct a constraint $op \in op_{defined}$, where op represents generated opcode of the instruction and $op_{defined}$ is the opcode specified in the specification. Accordingly, there is a constraint $op \in op_{illegal}$, where $op_{illegal}$ is the set of bytes that cannot be interpreted as an opcode.

Constraints on registers. We construct a constraint $reg \in reg_{context} \cup reg_{data}$, where reg represents generated register number. Meanwhile, the register numbers are classified under two different use sets as reg_{data} and $reg_{context}$. Specifically, reg_{data} is used to store the calculation result from the algorithm instruction, while $reg_{context}$ maintains the context information during the execution. The context information comes from the eBPF runtime and can also used to load dynamic memory address information. Since the eBPF runtime stores the context information to the register $r1$, we construct the default constraint $r1 \in reg_{context}$.

Constraints on values of operands and immediate arguments. The other constraints are decided by the value of operands and immediate arguments. Since we know the heap length and stack space of the eBPF runtime, we limit the possible memory address range under those predefined valid memory spaces as $\mathbb{M}$. We apply the memory address constraints to the operands of the memory instruction as $Imm \in \mathbb{M}$, where Imm denotes the generated value of operands.

After obtaining the constraints, BpfChecker automatically generates the initial IR program that satisfies the constraints. We create the initial IR program using two variations, as demonstrated below.

- Arithmetic-oriented. This mode explores states caused by arithmetic operations within the runtime environment. We increase

the generation weight of arithmetic operation instructions. When the candidate registers are overly abundant, the register and operand dependency among arithmetic instructions weakens. The real-world arithmetic-related bugs, such as CVE-2021-31440 shown in Figure 5, utilize operations of shifting, negation, multiplication, and addition on the same register to trigger boundary conditions of the tracked register states in the verifier. Furthermore, when all defined registers, namely `r0` to `r9`, are involved in arithmetic calculation during generation for 10 instructions, each register is utilized by the arithmetic instruction only twice on average. Since only a unique register state is considered a significant indicator during inconsistency detection, we restrict the number of available registers to half the total register count defined in the eBPF specification and limit the number of basic blocks generated.

- Control-flow-oriented. This mode aims to explore control-flow-related implementation in the eBPF runtime. In the verifier, since the typical control-flow analysis is performed on the control-flow graph, we prefer to generate additional basic blocks to augment the complexity of the nodes within the graph.

3.1.3 Semantic IR Corrector. The eBPF program generated from a given IR program can still violate the rules of the eBPF verifier or raise errors during the execution. This is due to the fact that the generator does not perform control-flow analysis on the IR program and can thus generate instruction with a dead loop. Performing precise control flow and value tracking introduces significant overhead and is impractical as it needs to be understood in detail for the internal implementation of each runtime. Identifying all the semantic checks performed in the verifier is a daunting job. Fortunately, we noticed that the occurrence of errors within the verifier invariably generates corresponding error messages, providing a semantic meaning of the check. The error message gives us a clear understanding of what is going on with the executed eBPF program. To improve the correctness of the generated program in a lightweight way, we introduce the semantic IR corrector to iteratively refine and mutate low-level eBPF programs based on the error messages captured during the differential execution.

The workflow of the semantic IR corrector is shown in Algorithm 1. Given an eBPF IR program, we extract error messages from the differential testing engine (Section 3.2) during execution. These messages indicate the invalidity of test cases and supplement instruction constraints and can be used to refine the test cases. More generally, we first identify the throw point of an error message, which is typically a verifier rejection or runtime error statement. Then, we analyze the message format and the invalid instruction location associated with the error message by the *extractErrorInstructionLocation* function. Specifically, as shown in Figure 6, while executing the memory load operator, we encounter an error message that complains about the access violation error. The error message comprises a formatted string including the exact invalid accessing address and the location (PC number) of the error instruction. We create a regular expression based on the error string format to extract the location of the error instruction and the error type. Afterward, we select the mutator from our predefined mutator pool according to the error type. The main error types we focused on are memory errors and division errors. For the other errors, we select the default mutator. We create three types of predefined mutators, as demonstrated below:

Algorithm 1 The Workflow of Semantic IR Corrector.

Input : P: Last executed eBPF IR program
Output: P': New eBPF IR program mutated by corrector

```
mutationTimes ← 0
P' ← P
while mutationTimes < T do
    E ← executeProgram(P)
    if E ≠ ∅ then
        L ← extractErrorInstructionLocation(E)
        S ← getMutationStrategyByErrorType(E)
        P' ← mutation(P, L, S)
    else
        break
    end
    mutationTimes ← mutationTimes + 1
end
```

- MemoryError Mutator. We first analyze whether the memory address of the erroneous instruction is loaded by the immediate value or the register according to the concrete opcode type. If the memory address is in the form of an immediate value, we replace the address with a random number from the predefined valid memory space $\mathbb{M}$. If the address is loaded from a register, we replace it with another random register in $reg_{context}$.
- DivisionError Mutator. Division errors occur when the divisor is zero. Since the generated instructions are constrained, we will not generate division instructions where the divisor is an immediate number 0. Therefore, the DivisionError implies that the divisor register is zero during execution. We replace the divisor register with a random register in $reg_{context}$ or insert an arithmetic instruction before the division instruction to change the value of the divisor register.
- Default Mutator. We randomly delete the erroneous instruction or replace it with a new instruction. The new instruction is either copied from the already executed instructions or generated by the constraint-based generator.

We use the selected mutator to mutate the given IR program and execute the newly mutated one on the differential testing engine within a limited time. When the new program does not raise the same error or the number of iterations reaches the boundary, we cease the corrector. Since we only perform mutations on the exact single instruction, the syntactic correctness of the test case is preserved.

3.2 Differential Testing Engine

We design a differential testing engine that can execute the eBPF programs on the eBPF runtimes and detect the differences between these runtimes. To identify the presence of a difference between two runtimes, we model the intermediate state of the runtime. After that, we initialize the runtimes to the same state before execution and collect the intermediate state that denotes the internal execution state of runtimes for comparison after the execution. Capturing

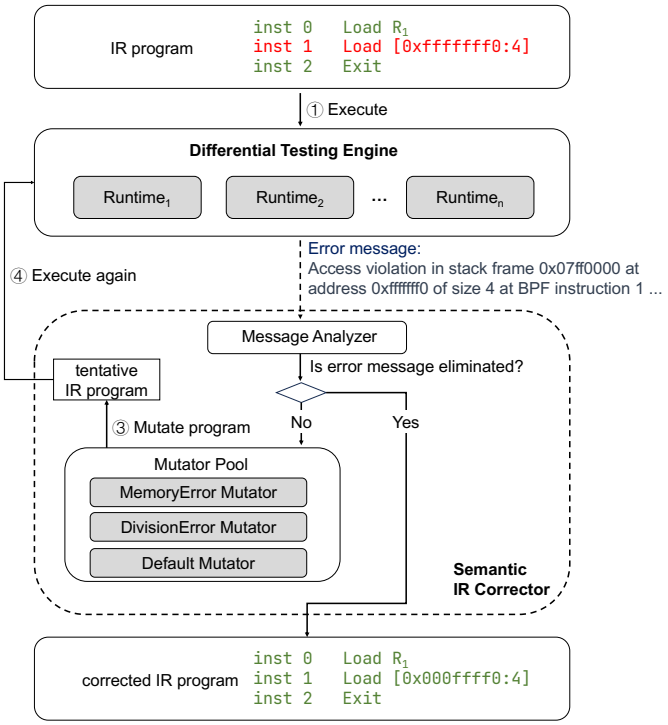

Figure 6: Example of how semantic IR corrector fixes the invalid eBPF program.

eBPF runtime differences through differential fuzzing can be formalized as follows. We use R to denote a specific eBPF runtime and P to denote the input eBPF program. The captured state after the execution can be represented as $S_E = R(P)$. For n eBPF runtimes under fuzzing, if any of the eBPF runtimes encounter an exception and throw error messages, we consider it as an invalid test case, which is beneficial for test case optimization. Furthermore, we identify an implementation flaw among R_i if there is a difference for n runtimes, as demonstrated below:

$$\forall i, j \in \{0, 1, \ldots, n\}, \quad \text{if } R_i(P) \neq R_j(P)$$

3.2.1 eBPF runtime model. We leverage the runtime variables to model the internal state of the runtime. Meanwhile, to guarantee the runtime has the same state before the given eBPF program is executed, we initialize these variables in different runtimes to the same before the execution. Similarly, we capture the difference among runtimes by comparing these variables. We define the initial runtime state as S_I, which is represented as a tuple $(Stack, Heap, R)$. Specifically, $Stack$ and $Heap$ refer to the stack and heap memory space in the eBPF runtime, which is allocated for each execution of the program. R denotes the register states before the execution. Meanwhile, we denote the runtime state after the execution by S_E, which is determined by the tuple $(Stack, Heap, R, E_{verifier}, E_{exec})$. The $Stack$, $Heap$ and R in S_E retain the same meanings as those in S_I. In addition, we use $E_{verifier}$ and E_{exec} to denote the error state in the eBPF runtime, including the verifier error state and the execution error state.

3.2.2 State analyzer. We need to set each runtime under testing to the same initial state S_I and capture the post-state S_E after the execution. Since all runtimes are fed with the same eBPF program as input, they are expected to have the same S_I. We introduce the following strategy to initialize the S_I.

Heap Since initializing a large-size heap memory is notably time-consuming, before the execution of the differential testing engine, we randomly generate the target heap memory size within 1024 KB. Subsequently, we generate initial memory data filled with random numbers based on the memory size. By fulfilling the randomly generated memory data, we can capture the differential memory access cases if inconsistent memory addresses are accessed.

Stack We ensure uniformity in the stack size across all runtimes. Since the stack content can not be controlled by the input eBPF program, we do not generate random stack memory content. Instead, we initialize the stack space with all-zero data.

R In the user-space eBPF runtime, all registers except for the context registers are initialized to zero by default. Due to the different registering mapping implementations in the JIT module among eBPF runtimes, instrumenting additional initialization code for the native register is imprecise and redundant. We generally initialize the register in the preface of the generated eBPF program. The instrumented preface code contains register load instructions (e.g., load zero to each register in reg_{data}) in a fixed basic block, which generally does not affect the program execution logic.

After executing the eBPF program, we dump and compare the post-state S_E by the following metric.

$E_{verifier}$ Since the divergence in design principles exists among different user-space eBPF verifiers, some verifiers tend to defer the soundness safety check to the runtime. While one invalid instruction is rejected by the verifier V_a, it can still be verified by another implementation of the eBPF verifier V_b. However, this doesn't necessarily mean that the implementation of the verifier V_b is incorrect. Instead, V_b assumes this invalid instruction will be detected during the runtime execution. If the program is verified by the verifier, the $E_{verifier}$ is defined as $\emptyset$. Therefore, it does not contain error-kind metadata. Otherwise, the detailed error metadata is extracted from the $E_{verifier}$. To reduce the false positive when comparing the different results from the verifiers, we only identify the verifier inconsistencies when all $E_{verifier}$ have error metadata with different types.

E_{exec} We instrument the runtime to capture the error state when the runtime raises errors during execution. Since some eBPF runtimes do not have limitations on the execution instruction count, we manually instrument code, which raises an error if certain numbers of instructions are executed. By instrumenting the instruction meter code, we align the execution limitation behavior among each runtime and prevent the runtime from entering an infinite loop.

3.2.3 Instrumentation. We instrument each runtime to set up the equivalent initial runtime state and capture the post-state after execution. The captured states $(Heap, Stack, R, E_{verifier}, E_{exec})$ are converted into a uniform format to align across different runtimes. Specifically, $(heap, Stack, R)$ are converted into a mapping structure

that includes the memory content and the 64-bit register value. Moreover, ($E_{verifier}$, E_{exec}) are converted into error types shared among different runtimes. As each runtime differs, the instrument requires a one-time engineering effort.

Runtime Modification: Initial State Configuration As each runtime has a distinct initialization method, we set up rBPF and vanilla rBPF via their built-in APIs. For Windows eBPF, we patch the initialization code for the *Stack*, *Heap*, and *R*. By writing the memory field of the internal variables, these states are re-initialized to ensure consistency with each other.

Runtime Modification: Execution State Probing To make a fine-grained oracle, we capture the intermediate states after each instruction execution in both JIT and interpreter modes. The JIT compiler of rBPF runtime generates machine code in a single pass over the bytecode and emits the machine code for each opcode without optimization. Therefore, we insert an additional trampoline during JIT compilation to capture the state after an instruction is executed. The trampoline for rBPF is implemented as follows:

- Save the register context to the stack.
- Invoke the rust callback function with an offset address. In the callback function, we map native registers to eBPF registers to capture the register state. Furthermore, by reading the fixed slot maintained and shared during the execution, we can capture the memory state $\mathbb{M}$.
- Restore the register context and rebalance the stack.

In interpreter mode, we collect the state after executing each opcode handler. Since each opcode handler is processed within a loop, the instrument code is inserted at the beginning of the loop. For Windows eBPF, we add the exception handler to convert the runtime exception into the error status. The instrument ensures that the number of states captured in both interpretive and JIT modes remains consistent.

4 EVALUATION

Subjects We select three mainstream user-space eBPF runtimes, namely Windows eBPF (version 0.13.0), vanilla rBPF (version 0.2.0), and Solana rBPF (version 0.7.0) as the subjects to evaluate our approach. We denote them as $eBPF_W$, $rBPF_{vanilla}$, and $rBPF_{Solana}$. Windows eBPF mainly consists of the PREVAIL verifier and the uBPF executor. Although Windows eBPF is compiled as a Windows kernel driver and runs in the kernel mode in production environments, we compile it as a user-space program to improve the efficiency of its execution and to facilitate the detection of its state. Since Solana rBPF is a fork of vanilla rBPF, we select both of them as the evaluation target to check whether the implementation flaws is introduced by the customized modification. Specifically, we individually capture the states from the JIT and interpreter mode for each user-space eBPF runtimes. Therefore, we have six states set after running on three eBPF runtimes with two modes. Note that due to the huge design disparities between the Linux kernel eBPF runtimes and the other runtimes, we do not generally apply it on the kernel eBPF runtimes and will discuss it in Section 5.2 with more details.

Implementation and settings. We implement BpfChecker with Python, C, C++, and Rust, to which the lightweight eBPF IR module makes substantial contributions. We compile each user-space eBPF runtime with the sanitizer feature enabled in the compilation options, including Address Sanitizer (ASan) [36] and Undefined Behavior Sanitizer (UBSan) [42]. This enables us to obtain detailed elucidations when an underlying memory issue occurs. All experiments were done on a server running Ubuntu 22.04 with an Intel Core i9-13900K processor and 128G DRAM. We repeat each experiment for 10 campaigns to mitigate the randomness by the fuzzing.

Baseline fuzzers. Since the existing state-of-the-art eBPF fuzzers (e.g., BRF and Buzzer) are not adapted to the user-space eBPF runtime and fail to comprehend the semantics of user-space eBPF runtime, they have mediocre performance in the user-space eBPF runtime. We further conduct an performance evaluation of the Buzzer, BRF, and BpfChecker against the baseline in Section 4.1. Consequently, we use the internal fuzzer inside each eBPF project as the baseline fuzzer. Specifically, we use [38] as the baseline fuzzer for Solana rBPF, and use [20] as the baseline fuzzer for Windows eBPF. Although vanilla rBPF does not have an internal fuzzer, its structure is similar to Solana rBPF. Consequently, we have adapted the fuzzer from Solana rBPF to vanilla rBPF, establishing it as the baseline. Notably, using these project fuzzers as the baseline doesn't noticeably degrade the baseline performance since they are actively maintained by the developer team and offer a more comprehensive understanding of instruction set semantics compared to state-of-the-art kernel eBPF fuzzers.

Research Questions. We conduct extensive experiments and analysis, aiming to answer the following research questions that are concerned with the effectiveness of BpfChecker and the root causes of the discovered implementation flaws.

Since the verifier rejects the invalid eBPF program, generating a high-quality eBPF program is imperative for bug detection. The effect of our eBPF IR design and semantic IR corrector is an essential step towards better understanding how test case optimization triggers bugs more effectively. Consequently, we study the following research question:

RQ1: Can BpfChecker generates representative test cases?

The capability of detecting implementation flaws demonstrates the effectiveness of our approach. Thus, we investigate the following research question:

RQ2: Is BpfChecker effective in bug finding?

Moreover, the bug symptoms and their root causes allow developers to gain more practical insights into addressing a bug. This leads us to study the following research question:

RQ3: What are the root causes of the detected bugs?

Table 1: The code coverage and semantic correctness rate (SCR) reached after 12-hour fuzzing by BpfChecker (denoted as *BC*), BpfChecker variants and baselines. The baselines differ for each subject because we use the internal fuzzer within each eBPF runtime.

Subject	Metric	Code Coverage				SCR			
		BC	BC_r	BC_i	Baseline	*BC*	BC_r	BC_i	Baseline
$rBPF_{Solana}$	Average	**63.1%**	48.9%	57.2%	45.2%	**21.3%**	15.9%	12.2%	9.6%
	Improvement	-	29.0%	10.3%	17.5%	-	34.0%	74.6%	121.9%
$rBPF_{vanilla}$	Average	**58.6%**	41.2%	54.1%	37.5%	**19.1%**	13.5%	12.6%	10.1%
	Improvement	-	42.2%	8.3%	56.3%	-	41.5%	51.6%	89.1%
$eBPF_W$	Average	**51.2%**	43.7%	48.0%	39.4%	**14.4%**	9.7%	8.3%	6.4%
	Improvement	-	17.2%	6.7%	29.9%	-	48.5%	73.5%	125%
Average (among subjects)		57.6%	44.6%	53.1%	40.7%	18.3%	13.0%	11.0%	8.7%

Table 2: The code coverage and semantic correctness rate (SCR) reached after 12-hour fuzzing by BpfChecker (denoted as *BC*), BRF, Buzzer and baselines. The baselines differ for each subject.

Subject	Code Coverage				SCR			
	BC	BRF	Buzzer	Baseline	BC	BRF	Buzzer	Baseline
$rBPF_{Solana}$	**63.1%**	<5%	32.1%	45.2%	**21.3%**	<0.1%	1.2%	9.6%
$rBPF_{vanilla}$	**58.6%**	<5%	26.3%	37.5%	**19.1%**	<0.1%	1.5%	10.1%
$eBPF_W$	**51.2%**	<5%	22.0%	39.4%	**14.4%**	<0.1%	2.3%	6.4%

Table 3: Covered instruction types (CIT) reached after 12-hour fuzzing by BpfChecker and baselines. The baselines differ for each subject.

Subject	*BC*	Baseline	Improvement
$rBPF_{Solana}$	121	109	11.0%
$rBPF_{vanilla}$	108	95	13.7%
$eBPF_W$	96	87	10.3%
Average	108	97	11.3%

Table 4: The average number of executed test cases per second over a 12-hour fuzzing by BpfChecker and baselines. The baselines differ for each subject.

Subject	BpfChecker	Baseline	Slowdown
$rBPF_{Solana}$	650	880	26.1%
$rBPF_{vanilla}$	580	790	26.6%
$eBPF_W$	2610	5960	56.2%
Average	1280	2543	49.7%

4.1 Effectiveness of Test Case Generator (RQ1)

BpfChecker generates the test case based on the eBPF IR, which complies with the BPF instruction specification from the eBPF standardization. To demonstrate the effectiveness of the test case generator, we measure the covered instruction types, the semantic correct rate, and the code coverage during the fuzzing campaign. To demonstrate the effectiveness of the lightweight eBPF IR and the semantic IR corrector , we conduct an ablation study. We create variants of BpfChecker by disabling the semantic IR corrector (BC_i) and replacing the IR program generator with a random one (BC_r). We compare the performance of these variants and the vanilla BpfChecker on two metrics, including the instruction type coverage and the semantic correctness rate of the generated programs. Specifically, we calibrate the instruction type coverage and the semantic correctness on the interpreter mode of each eBPF runtime. For a correctly implemented runtime, these data should be the same under the JIT and interpreter modes.

Semantic correctness rate. Generating syntactically and semantically correct test cases is essential to explore the deep area of the eBPF runtime. The semantic correctness rate (SCR) is the percentage of semantically correct test cases, i.e., those validated by the eBPF verifier, among all samples generated over a specified duration. A higher SCR indicates the capability to explore more intricate states and uncover more profound bugs. We sample 100,000 programs generated by each variant on all subjects, and compute the number of semantically correct samples. We repeat the experiment for 10 campaigns. The average SCR value for each variant is reported in Table 1.

According to the column "SCR" in Table 1, only 8.7% programs generated by baseline fuzzers are semantically correct, which means all the others can contain illegal instructions and they are not effective in triggering the potential inconsistencies among runtimes. Conversely, BpfChecker achieves an 18.3% semantic correctness rate on average, surpassing the baseline by more than double. Compared with BpfChecker and BC_i, semantic IR corrector reveals an improvement in SCR of up to 74.6%. Even in the worst case, semantic IR corrector improves the SCR by 51.6% on $rBPF_{vanilla}$. Moreover, the removal of the IR (BC_r) results in a reduction of at least 34% in the SCR. The significant SCR improvement indicates that both IR and semantic IR corrector can effectively improve the generation effectiveness.

Code coverage. We measure the code coverage after 12-hour fuzzing by BpfChecker and the baseline, repeating for 10 campaigns. The average code coverage for each subject is reported in Table 1. Compared with the coverage reached by the baselines, BpfChecker achieves up to 56.3% improvement. We observe that the code coverage is significantly decreased when the IR program generator is disabled (BC_r). In the worst case, the coverage of

$rBPF_{vanilla}$ drops by 42.2% without the IR program generator. The removal of semantic IR corrector decreases the coverage by at least 6.7%, revealing that the quality of testcases affects the coverage.

Covered instruction types. For each testing subject, we measure the number of covered instruction types (CIT) reached after 12 hours of fuzzing. We only execute the verified programs and count the executed instruction types by calculating the joint interpreter code coverage related to the instruction handler. We leverage the CIT to understand whether the generated program itself can be fully explored and executed by the runtime.

The results of CIT are reported in Table 3. Among verified programs, more than 10% types of instructions cannot be covered by the random generator, resulting in relatively limited implementation exploration. Conversely, BpfChecker achieves 11 more covered instruction types and can thus reveal implementation flaws that the baselines fail.

Kernel eBPF fuzzer evaluation. State-of-the-art kernel eBPF fuzzers, such as Buzzer [9] and BRF [11], demonstrate effectiveness in identifying flaws within the kernel verifier and the kernel eBPF API. However, they lack the semantic understanding of the user-space eBPF instruction set and do not model user-space eBPF. Specifically, BRF primarily generates programs targeted at the kernel eBPF API, and it is based on source code compilation, unable to generate or mutate on low-level instructions. Consequently, their performance in detecting faults in user-space eBPF is mediocre. Therefore, we used the project fuzzer as a baseline to highlight the performance of the state-of-the-art fuzzers.

We evaluate the effectiveness of Buzzer and BRF by replacing the code generation component of BpfChecker with the respective code generators from Buzzer and BRF. We use Buzzer in commit hash *39b2b25* and BRF in commit hash *047d1c9*. We employ the same machine environment as utilized in the evaluation (Section 4). The experiment is repeated 10 times.

The average code coverage and SCR for each subject are reported in Table 2. The majority of programs generated by BRF primarily consist of kernel eBPF API calls, which are prone to errors when executed in user-space eBPF. This leads to significantly low code coverage and reduced semantic accuracy. Buzzer leverages a templated approach to produce eBPF programs, yet it lacks semantic awareness to the instruction set of user-space eBPF runtime. Additionally, it has limited program generation strategy and fails to generate sufficient instruction types, resulting in notably low code coverage. Using the instruction-level program corrector and effective IR, BpfChecker achieves more than 10 × semantic correctness and 2 × code coverage of the Buzzer. Additionally, during the 24-hour fuzzing campaign, both Buzzer and BRF failed to find any bugs that BpfChecker could identify within 12 hours.

As Buzzer uses template-based program generation, its semantic accuracy for generated programs is significantly lower compared to BpfChecker. Since BRF generates API-oriented eBPF programs and relies solely on source code-level mutations, it fails to produce a diverse range of low-level instructions, which results in significantly lower semantic accuracy in user-space eBPF. The project fuzzer, which accounts for the subtle semantic variations in its instruction set, is more representative. Moreover, BpfChecker significantly outperforms state-of-the-art fuzzers in terms of program correctness and effectiveness in detecting implementation flaws in user-space eBPF runtime.

Answer to RQ1: BpfChecker can generate sufficient test cases which are all syntactical correct instructions and can reach **18.3%** semantic correctness rate on average, which is **2.1x** of the baselines on average for the evaluated targets. In terms of code coverage, BpfChecker achieves an improvement of at least 17.5% compared to the baselines. Furthermore, BpfChecker can explore all types of instructions implemented in the eBPF runtimes, while 11 types of instruction cannot be explored by the baselines.

4.2 Effectiveness of BpfChecker(RQ2)

4.2.1 Throughput. BpfChecker generates eBPF programs using the constraint-based generator mentioned in Section 3.1.2 under error message guidance. In this section, we evaluate the performance of the BpfChecker in terms of the execution speed.

The evaluation approach is as follows. For a given test case, BpfChecker executes them on a single runtime in both JIT and interpreter mode. We run the BpfChecker and the baseline fuzzer of both Solana rBPF and Windows eBPF project. To mitigate the nondeterministic factors introduced by multithreading execution, all tools are running on a single CPU core and is repeated for 10 times. Moreover, we measure the average number of test cases executed per second over a 12-hour period.

The evaluation result shown in Table 4 demonstrates the average executed test cases for both BpfChecker and the baseline fuzzer. For rBPF runtime, the average slowdown of BpfChecker compared to the baseline fuzzer is 26.1%. For Windows eBPF, since libFuzzer leverages in-process execution, it is faster than BpfChecker which executes the standalone eBPF runtime binary. However, considering the benefits of triggering and detecting implementation flaws, the slowdown is acceptable. Even if the overall slowdown reached 49.7%, BpfChecker can still detect most of the implementation flaws that the baseline fuzzers fail to identify, which is demonstrated in the Section 4.2.2.

4.2.2 Implementation Flaws Discovery. We conduct differential fuzzing on these eBPF runtimes and find a total of *28* new implementation flaws as described in Table 5, including *7* crash and *21* non-crash implementation flaws. We also obtained 2 CVEs for critical vulnerabilities. We triage and confirm these bugs through manual efforts. Notably, the baseline fuzzer for rBPF is written on the top of the cargo-fuzz, while on the Windows eBPF, it is written on the top of the libFuzzer. We denote both of the fuzzer as the *Baseline* column shown in Table 5. Among the 28 new bugs, only 3 bugs can be found by the baseline fuzzer after running on each eBPF runtime with the vulnerable version for 7 days. However, BpfChecker can find all these bugs on the vulnerable eBPF runtime within 12 hours, showing its effectiveness in bug detection. The bugs found by the baseline fuzzer are all memory corruption bugs in the Windows eBPF verifier, showing that the existing oracle

Table 5: Unique bugs found by BpfChecker over the course of multiple months. All identified bugs are reported to the respective vendor, and we actively work with the developers to fix the vulnerabilities. All details concerning the bugs and links to their fixes are contained within the repository https://github.com/bpfchecker/BpfCheckerSource. The bug ids listed in the *ID* column are assigned by PREVAIL verifier issues list [45] (⊙), vanilla rBPF issues list [33] (⊕), Solana rBPF issues list [37] (÷), Solana security advisories [39] (∗), uBPF issues list [31] (⊛), and Microsoft Security Response Center [19] (○). The *Module* column indicates the component of the respective eBPF runtime where bugs manifest. The *Baseline* column indicates whether the bug can be found through the project's built-in fuzzer.

Subject	ID	Module	Type	Status	Baseline
$rBPF_{Solana}$	CVE-2022-23066	JIT	Correctness	Fixed	✗
$rBPF_{Solana}$	CVE-2021-46102	Parser	Correctness	Fixed	✗
$rBPF_{Solana}$	3qwj-hrj9-6x95*	Parser	Correctness	Fixed	✗
$rBPF_{Solana}$	87mf-r7qw-cgqg*	Interpreter	Correctness	Confirmed	✗
$rBPF_{Solana}$	frh9-6jf9-7c63*	Interpreter	Correctness	Duplicated †	✗
$rBPF_{Solana}$	q9wq-j357-4gcv*	JIT	Correctness	Fixed	✗
$rBPF_{Solana}$	270$^{\div}$	JIT	Correctness	Fixed	✗
$rBPF_{Solana}$	269$^{\div}$	Verifier	DoS	Fixed	✗
$rBPF_{Solana}$	549$^{\div}$	Verifier	Correctness	Fixed	✗
$rBPF_{vanilla}$	91$^{\oplus}$	Disassembler	Correctness	Fixed	✗
$rBPF_{vanilla}$	92$^{\oplus}$	Disassembler	Correctness	Fixed	✗
$rBPF_{vanilla}$	94$^{\oplus}$	Interpreter	Correctness	Fixed	✗
$rBPF_{vanilla}$	95$^{\oplus}$	Interpreter	Correctness	Fixed	✗
$rBPF_{vanilla}$	96$^{\oplus}$	Interpreter	Correctness	Fixed	✗
$rBPF_{vanilla}$	99$^{\oplus}$	Interpreter	Correctness	Fixed	✗
$rBPF_{vanilla}$	101$^{\oplus}$	Interpreter	Correctness	Reported	✗
$eBPF_{W}$	068772°	JIT	Memory Corruption	Fixed	✗
$eBPF_{W}$	068780°	Interpreter	Correctness	Fixed	✗
$eBPF_{W}$	068726°	Verifier	DoS	Confirmed	✗
$eBPF_{W}$	068574°	Verifier	Memory Corruption	Fixed	✓
$eBPF_{W}$	067989°	Verifier	Memory Corruption	Fixed	✓
$eBPF_{W}$	243$^{\odot}$	Verifier	Correctness	Fixed	✗
$eBPF_{W}$	545$^{\odot}$	Verifier	DoS	Reported	✗
$eBPF_{W}$	433$^{\circledast}$	Interpreter	Correctness	Reported	✗
$eBPF_{W}$	434$^{\circledast}$	Interpreter	Memory Corruption	Reported	✓
$eBPF_{W}$	435$^{\circledast}$	JIT	Correctness	Reported	✗
$eBPF_{W}$	436$^{\circledast}$	JIT	Correctness	Reported	✗
$eBPF_{W}$	437$^{\circledast}$	JIT	Correctness	Reported	✗

† Public discovery of a vendor-concealed Bug.

cannot detect subtle correctness bugs. Moreover, the baseline fuzzer fails to find any bugs that cannot be detected by BpfChecker.

Answer to RQ2: BpfChecker can identify implementation flaws among each component of the eBPF runtime. The existing state-of-the-art fuzzers fail to uncover 25 out of the 28 implementation flaws found by BpfChecker even if their execution speed surpasses that of the BpfChecker. Bug oracles of the existing work are blind to these implementation flaws despite an input program being able to produce faulty samples.

4.3 Analysis of implementation flaws(RQ3)

We summarize the implementation flaws discovered by BpfChecker during our evaluation over three eBPF runtimes. The buggy implementations occur 7 times within the verifier, 7 times within the JIT, and 11 times within the interpreter. We present the case study of four typical vulnerabilities detected by the BpfChecker. Those vulnerabilities fulfill each part of the eBPF runtime, including the verifier, JIT compiler, and interpreter. Notably, CVE-2022-23066 is a miscalculation vulnerability in the rBPF interpreter, caused by the lack of sign extension to the register. CVE-2021-46102 is a vulnerability in the rBPF parser that causes an overflow in address calculation during parsing. Both of these vulnerabilities are of high severity and have received bounties from the vendor.

Case Study: miscalculation in rBPF interpreter. Figure 7a is a simplified PoC triggering inconsistent results between JIT and interpreter in Solana rBPF. The bug is caused by an incorrect sign bit extension when calculating the *MUL* instruction. The bug does not cause any crashes and is hard to be detected by other fuzzers. During the calculation of the interpreter for *MUL*32 instruction, it directly cast the result of a signed 32-bit result to a signed 64-bit number and

```
1 add32 r0, 0x1000ffff // r0=0x1000ffff
2 mul32 r0, r0         // rBPF interpreter: r0=0xffffffffdffe0001
                       // others:           r0=0x00000000dffe0001
3 exit
```

(a) Miscalculation bug in rBPF interpreter.

```
1 mov r5, 0x1               // r5=1
2 ldxdw r5, [r10 - 0x6650]  // load value from stack
                            // Windows eBPF: no exception raised
                            // others: return AccessViolation error
3 exit
```

(b) Incorrect boundary check in the verifier of Windows eBPF.

```
1 mov64 r8, 0x054545ff
2 lsh64 r8, r8         // vanilla rBPF: panic with overflow shift
                       // others:       r8=0x8000000000000000
3 exit
```

(c) Incorrect shift implementation in rBPF interpreter.

```
1 mov64 r0, 0
2 mov64 r8, r4
3 if r6 > -1079394508 goto +12343
4 mov64 r1, r0
5 add32 w8, w4
6 add32 w4, w4
7 mov64 r8, 207967935
8 if r0 >= 1459617637 goto -5
9 if w0 != 207050400 goto +24936
10 goto -5         // verifier of eBPF Windows: trapped into an infinite loop
                   // other: raise runtime PC jump outside error
```

(d) Incorrect control-flow analysis in Windows eBPF leading to the infinite loop during verification.

Figure 7: The proof-of-concept eBPF programs of the demonstrated implementation flaws. The comment of each program demonstrates the simplified execution state.

implicitly performed the sign-bit extension. If the 32-bit representation of the calculated result is negative, the sign-extended 64-bit representation will be extended as negative, i.e., the upper 32-bits are populated with ones. However, the eBPF specification stipulates that the upper bits following 32-bit integer operations should be zeroes. The initial value of $r0$ is set to 0x1000ffff by the *ADD*32 instruction. In the *MUL*32 instruction, the self-multiplication result should be 0xdffe0001. Due to the wrong sign extension, the interpreter sets the $v0$ to the wrong 0xffffffffdffe0001. The wrong result further breaks the intended behavior between the high-level contract language and the low-level bytecode execution, leading to the consensus issue in the blockchain network. BpfChecker captures this inconsistency bug through the register comparison rules.

Case Study: incorrect boundary check in Windows eBPF verifier. Figure 7b is a simplified PoC triggering heap buffer overflow in the verifier of Windows eBPF. In the forward analyzer of the verifier, it traverses and visits all the eBPF instructions and simulates the load stack operation if the current instruction is a *LoadStack* instruction (i.e., read memory on $r10$ register with offset). It finally queries whether the target stack memory with the offset k is a number type or not. The k value is read directly from the user-provided offset without verification. Therefore, when accessing the simulated stack array with the unsanitized k, the offset can be larger than the array size `EBPF_STACK_SIZE`, leading to the heap buffer overflow. This bug can be leveraged to read out-of-boundary addresses in kernel memory, as the controlled k value ranges from `0` to `SIZE_T_MAX`.

Case Study: incorrect shift implementation in rBPF Interpreter. Figure 7c demonstrates a PoC program that triggers undefined behaviours in the interpreter of vanilla rBPF. The eBPF specification specifies that "Shift operations use a mask of 0x3F (63) for 64-bit operations and 0x1F (31) for 32-bit operations". However, the interpreter in $rBPF_{vanilla}$ lacks the mask and is not compliant with the specification. The source bits will overflow when the shift operation is performed on a large offset like 0x054545ff, as shown in the PoC program. This leads to undefined behaviour since the low-level machine codes on different architectures have different

implementations when the shift operation overflows. Hence, the calculated results vary across machines with different architectures.

Case Study: incorrect control-flow analysis in Windows eBPF verifier. The eBPF program shown in Figure 7d can trap the Windows eBPF verifier in the infinite loop. Specifically, when there are more than two back edges inside the CFG and the infinite loop exists in the eBPF program, the Windows eBPF verifier follows the loop flow of the program and doesn't properly check the finitude during simulation.

Design Disparities Furthermore, during our fuzzing campaigns spanning several months, we find several design disparities in different runtimes, leading to the initial false positive cases. These cases are valid and reproducible, yet they all fall within the *runtime-specific design specifications*. We categorize these inconsistencies caused by the design disparities and introduce the following rules to eliminate false positive cases.

- *Stochastic runtime address* The runtime heap and stack address are not fixed and are allocated dynamically each time the program is executed. Besides, user-space eBPF runtimes allow those stack memory addresses to be accessed. The concrete host address stored in the registers causes false positives due to the inconsistent register states. When we generate a program using those host-related registers, we only compare the error message from the verifier and executor since the concrete host address can taint registers.
- *Division by zero* The eBPF specification considers division by zero as a regular behavior, whereupon encountering a divisor of zero, the target register is set to zero. Conversely, within the design of Solana rBPF, division by zero triggers the *DivisionByZero* error. We ignore the state comparison when the execution result from Solana rBPF is *DivisionByZero*. This exemption will not lead to additional false negatives since the other part of the program logic can be tested when we generate a semantic identical program without such division instruction.
- *Instruction measurement* The eBPF specification allows a maximum of 1 million instructions to be executed at runtime, and the program size is limited to only 4096 instructions. All verifiers in user-space eBPF runtimes do not perform sophisticated loop analysis or check the program's termination. Moreover, the virtual machines of vanilla rBPF and Windows eBPF do not control the number of instructions or the program, leading to the runtimes getting trapped in infinite loops if the program is infinite. Since Solana rBPF limits the executed instruction during execution, we leverage the instruction meter in the Solana rBPF to measure the instruction executed count and determine whether the program is infinite. If the program execution on Solana rBPF raises the *ExceededMaxInstructions*, we will not execute this program on other runtimes to avoid unnecessary program hanging. This optimization enhances the throughput of fuzzing since the runtime will not be subject to program-induced infinite loops.
- *Error handling mechanisms* The eBPF specification asserts that runtime should not experience crashes due to abnormal behavior of the program. Runtime errors need to be handled judiciously by the host. However, the JIT compiler in vanilla rBPF does not implement the proper error-handling mechanisms, resulting in execution errors that can crash runtime directly. Such crashes are ultimately captured by the host system, affecting the robustness of the runtime and resulting in the differential testing engine's inability to capture execution results accurately.
- *Configuration* Some differences are due to the inconsistent configurations of the runtimes. For example, the stack frame size in SBF V1 implementation of Solana rBPF is 4096, while the size in Windows eBPF is 512. Therefore, given a program accessing memory with a stack space offset of 1000, the execution in Solana rBPF will not cause errors, while in Windows eBPF, the execution will raise access violation errors. Moreover, Solana rBPF set the initial value of register $r1$ to zero, while vanilla rBPF set it as the host heap address under default configuration.

Answer to RQ3: During the fuzzing campaign, we identify and categorize 5 types of root causes for inconsistencies related to designed disparities. The implementation flaws exist in each component of the eBPF runtimes, yet the execution component has been overlooked by the existing work. In addition, as shown in our case studies, implementation flaws generally do not manifest as segmentation faults or sanitizer violations, while they have a severe impact on the entire system.

5 DISCUSSION AND LIMITATION

5.1 Suggestions for developers.

We summarize two concrete suggestions for eBPF runtime developers according to the analysis of the implementation flaws. ❶ Beware of the subtle differences in data types for the arithmetic instructions. Based on our findings, handlers for data types at boundary conditions can easily be overlooked. It is recommended that the handler of special cases for each data type be reviewed when implementing the eBPF specification. ❷ Enhancing test suites by comparing the results between the JIT and interpreter. In some user-space eBPF runtimes (e.g., Windows eBPF), test cases often only validate the results of either the JIT or the interpreter individually, neglecting the comparison of consistency between them. This oversight makes existing test suites fail to capture correctness issues during development.

5.2 Limitation

Limitations due to the fixed IR constraint rules. BpfChecker generates the eBPF program under the predefined constraints. However, as the eBPF runtime evolves, the rules of the verifier will be updated, and new instruction types will be added for the executor. To ensure the quality of the generated program, additional modifications need to be made to the lightweight IR, along with the inclusion of new constraints. Although adapting the updated eBPF instruction sets and generating succinct eBPF instructions can be achieved by enhancing LLVM's eBPF backend, it requires a certain engineering effort. In the future, we aim to utilize more powerful NLP techniques to automatically adapt updated specifications to

our lightweight IR and generate the corresponding constraints for the new instruction types.

Limitations due to the design disparities. Since the kernel verifier utilizes safety rules that are not implemented by the user-space eBPF verifier, their validation results for the same program often diverge. For instance, given an eBPF program that reads data from the stack before writing into it, the kernel eBPF verifier would reject it since the uninitialized stack access is not allowed. However, in user-space eBPF runtime (e.g., rBPF, Windows eBPF), the stack has been initialized to zero; hence, the verifier does not check the uninitialized memory access. The design disparities further lead to undesirable false positives that do not manifest as real bugs. Since completely eliminating the false positives caused by the design disparities is challenging, we do not generally apply the BpfChecker on the kernel eBPF framework. In the future, we plan to explore how to generate a subset of programs that adhere to the common standards of kernel and user-space runtime.

6 RELATED WORK

eBPF Fuzzing. Fuzzing is an effective approach to detecting bugs, and many works in this area are relevant to our work. Specifically, Syzkaller [7], the state-of-the-art kernel fuzzer, is capable of testing the eBPF framework by generating random bpf() system calls. It has been integrated into the kernel upstream to test continuously and has demonstrated promising capability in uncovering memory bugs. Another work [30] proposed by IO Visor Project utilizes the LLVM framework to test the eBPF verifier from kernel space to user space. To do so, it replaces related kernel routines with simplified user-space versions and performs coverage-guided fuzzing by libfuzzer. Benjamin et al. [28] proposed a syntax-aware fuzzer targeting the eBPF subsystem in the Linux kernel based on Angora [3]. It uses the sample eBPF programs in the Linux kernel source tree as the initial corpus and uses gradient descent to guide the mutation. However, the inputs generated by these tools are likely to have invalid instructions, which will be rejected by the verifier early. Moreover, even if an eBPF program that passes the verifier's checks is generated, detecting correctness bugs is challenging for these tools since they all use the sanitizer as their sole oracle. Buzzer [9] is a recent work that targets the verifier. It randomly generates eBPF programs, which mainly involve simple ALU and JMP instructions, thereby not extensively probing the verifier's intricate checking mechanisms. Moreover, it failed to capture correctness issues in the other components of the eBPF runtimes. As a result, Buzzer only found bugs within the verifier after an extended duration of testing.

The work proposed by Scannell [35] is a bytecode-level semantic-aware fuzzer that targets the JIT compiler. It generates the bytecode incorporating a degree of semantics of the register states. If the generated program is verified by the verifier, it assumes that the map-writing operation inside the program is always within bounds, and thus, faulty pointer arithmetic is detected when the value of the map remains unchanged after the program is executed. The experiments show a mere 0.77% validity rate among the generated eBPF programs due to the limited awareness of semantics. *Unlike the above studies,* BpfChecker *considers the consistency of the key states in the runtime as the semantic oracle and utilizes the semantic-aware generator to generate high-quality test cases.*

Differential Testing. Differential testing is a powerful dynamic technique introduced by McKeeman et al. [16] to identify bugs or inconsistencies between software systems implementing the same functionality. Deng et al. [6] introduced a differential testing framework of cross deep learning framework. It utilizes a joint API test across multiple frameworks to construct more fine-grained constraints and identify inconsistencies more efficiently. However, since the user-space eBPF framework has new API for the program execution, it is unsuitable to perform API-based testing on the user-space eBPF runtime. Examiner [12] and WADIFF [54] leverage a symbolic execution engine for architecture specification language (ASL) to generate input test cases for their targeted system. However, the ASL cannot be leveraged to describe the semantics of the eBPF program. Csmith [51] generates random C programs with complex syntactic structures and performs differential testing on the compile result from the C compiler. However, these frameworks can not be applied to eBPF directly due to the absence of prior knowledge regarding the state of eBPF runtime. *Apart from the above studies,* BpfChecker *is the first general differential fuzzing framework that uncovers implementation flaws in the eBPF runtimes.*

Differential Analysis. Differential analysis usually adopts hybrid techniques to strengthen the bug detection capability. ParDiff [52] uses static differential analysis to detect domain-specific bugs of the network protocol parsers. It firstly extracts finite state machines from programs to represent protocol format specifications. Subsequently, it leverages bisimulation and SMT solvers to detect fine-grained and semantic inconsistencies between the network protocol parsers. However, due to the challenge of understanding and extracting the code generation process of the JIT compiler, it will take too much effort to analyze JIT compilers of the eBPF runtimes. *Different from the existing differential analysis,* BpfChecker *employs concrete execution to address the challenges of state abstraction.*

7 CONCLUSION

In this paper, we present BpfChecker, a differential fuzzing framework that can satisfy the semantics required by the eBPF runtime to detect implementation flaws in the eBPF runtimes. To address the oracle challenges posed by the nature of the correctness bug, BpfChecker uses the differential testing engine to identify any unexpected execution discrepancy caused by the implementation of the eBPF runtime. Furthermore, BpfChecker uses the lightweight eBPF IR and the semantic IR corrector to enhance the quality of the test cases. BpfChecker outperforms state-of-the-art fuzzing approaches in terms of both semantic correctness rate and identified bug finding. Simply put, we have uncovered a total of 28 new bugs, with 2 of them assigned CVE numbers and received $800,000 bounty from the vendor. In addition, we analyze the root causes and potential impacts of these vulnerabilities in a real-world scenario.

ACKNOWLEDGMENTS

We thank all anonymous reviewers for their invaluable comments. This work is partially supported by the National Key R&D Program of China (No. 2022YFE0113200), the National Natural Science Foundation of China (NSFC) under Grant 62172360, U21A20464, the

Hong Kong ITF Project (No. PRP/005/23FX). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of funding agencies.

REFERENCES

[1] Marcelo Abranches, Oliver Michel, Eric Keller, and Stefan Schmid. 2021. Efficient network monitoring applications in the kernel with ebpf and xdp. In *2021 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, 28–34.

[2] Maher Alharby, Amjad Aldweesh, and Aad Van Moorsel. 2018. Blockchain-based smart contracts: A systematic mapping study of academic research (2018). In *2018 International Conference on Cloud Computing, Big Data and Blockchain (ICCBB)*. IEEE, 1–6.

[3] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 711–725.

[4] Alexei Starovoitov Daniel Borkmann. 2024. *eBPF*. https://ebpf.io

[5] Poorna Gaddehosur Dave Thaler. 2021. *Making eBPF work on Windows*. Retrieved March 1, 2024 from https://cloudblogs.microsoft.com/opensource/2021/05/10/making-ebpf-work-on-windows

[6] Zizhuang Deng, Guozhu Meng, Kai Chen, Tong Liu, Lu Xiang, and Chunyang Chen. 2023. Differential Testing of Cross Deep Learning Framework {APIs}: Revealing Inconsistencies and Vulnerabilities. In *32nd USENIX Security Symposium (USENIX Security 23)*. 7393–7410.

[7] Andrey Konovalov. Dmitry Vyukov. 2024. *Syzkaller: an unsupervised coverage-guided kernel fuzzer*. https://github.com/google/syzkaller

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

[9] Google. 2024. *Buzzer - An eBPF Fuzzer toolchain*. https://github.com/google/buzzer

[10] Alex Groce, Mohammad Amin Alipour, and Rahul Gopinath. 2014. Coverage and its discontents. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. 255–268.

[11] Hsin-Wei Hung and Ardalan Amiri Sani. 2023. BRF: eBPF Runtime Fuzzer. *arXiv preprint arXiv:2305.08782* (2023).

[12] Muhui Jiang, Tianyi Xu, Yajin Zhou, Yufeng Hu, Ming Zhong, Lei Wu, Xiapu Luo, and Kui Ren. 2022. EXAMINER: Automatically locating inconsistent instructions between real devices and CPU emulators for ARM. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 846–858.

[13] Linux kernel. 2024. *eBPF verifier — The Linux Kernel documentation*. Retrieved March 1, 2024 from https://docs.kernel.org/bpf/verifier.html

[14] Jung-Bok Lee, Tae-Hee Yoo, Eo-Hyung Lee, Byeong-Ha Hwang, Sung-Won Ahn, and Choong-Hee Cho. 2021. High-performance software load balancer for cloud-native architecture. *IEEE Access* 9 (2021), 123704–123716.

[15] Youlin Li, Weina Niu, Yukun Zhu, Jiacheng Gong, Beibei Li, and Xiaosong Zhang. 2023. Fuzzing Logical Bugs in eBPF Verifier with Bound-Violation Indicator. In *ICC 2023-IEEE International Conference on Communications*. IEEE, 753–758.

[16] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[17] Microsoft. 2024. *eBPF distributed networking observability tool for Kubernetes*. Retrieved March 1, 2024 from https://github.com/microsoft/retina

[18] Microsoft. 2024. *eBPF implementation that runs on top of Windows*. https://github.com/microsoft/ebpf-for-windows

[19] Microsoft. 2024. *Microsoft Security Response Center*. https://msrc.microsoft.com

[20] Microsoft. 2024. *Windows eBPF project fuzzer*. https://github.com/microsoft/ebpf-for-windows/tree/b9d6cb6b7edcc5314413d866a63d36ebc41ab14d/tests/libfuzzer

[21] MITRE. 2020. *CVE-2020-8835*. Retrieved March 1, 2024 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8835

[22] MITRE. 2022. *CVE-2022-23066*. Retrieved March 1, 2024 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-23066

[23] MITRE. 2023. *CVE - CVE-2023-2163*. Retrieved March 1, 2024 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-2163

[24] MITRE. 2024. *CVE - CVE-2021-46102*. Retrieved March 1, 2024 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=2021-46102

[25] MITRE. 2024. *CVE - CVE-2024-26588*. Retrieved March 1, 2024 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2024-26588

[26] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. 2019. Scaling symbolic evaluation for automated verification of systems code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 225–242.

[27] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. 2020. Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the linux kernel. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 41–61.

[28] Benjamin Curt Nilsen. 2020. *Fuzzing the Berkeley Packet Filter*. University of California, Davis.

[29] Hui Peng, Zhihao Yao, Ardalan Amiri Sani, Dave Jing Tian, and Mathias Payer. 2023. GLeeFuzz: Fuzzing WebGL Through Error Message Guided Mutation. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1883–1899.

[30] IO Visor Project. 2019. *eBPF fuzzing framework based on libfuzzer and clang sanitizer*. https://github.com/iovisor/bpf-fuzzer

[31] IO Visor Project. 2024. *uBPF issue list*. https://github.com/iovisor/ubpf/issues

[32] IO Visor Project. 2024. *Userspace eBPF VM*. https://github.com/iovisor/ubpf

[33] Qmonnet. 2024. *rBPF issues list*. https://github.com/qmonnet/rbpf/issues

[34] Qmonnet. 2024. *Rust Virtual Machine and Jit Compiler for eBPF programs*. https://github.com/qmonnet/rbpf

[35] Simon Scannell. 2021. *Fuzzing for eBPF JIT bugs in the Linux kernel*. https://scannell.io/posts/ebpf-fuzzing

[36] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*. 309–318.

[37] Solana. 2024. *Solana rBPF issues list*. https://github.com/solana-labs/rbpf/issues

[38] Solana. 2024. *Solana rBPF project fuzzer*. https://github.com/solana-labs/rbpf/blob/f3758ecee89198433422f751beee7f0f52dbcd55/src/fuzz.rs

[39] Solana. 2024. *Solana Security Advisories*. https://github.com/solana-labs/solana/security/advisories

[40] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. 2001. A dynamic optimization framework for a Java just-in-time compiler. *ACM SIGPLAN Notices* 36, 11 (2001), 180–195.

[41] 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. (2024).

[42] The Clang Team. 2024. *Undefined Behavior Sanitizer*. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[43] V8 Team. 2023. *V8's Fastest Optimizing JIT*. https://v8.dev/blog/maglev

[44] Vbpf. 2024. *eBPF verifier based on abstract interpretation*. https://github.com/vbpf/ebpf-verifier

[45] Vbpf. 2024. *PREVAIL eBPF verifier issue list*. https://github.com/vbpf/ebpf-verifier/issues

[46] Marcos AM Vieira, Matheus S Castanho, Racyus DG Pacífico, Elerson RS Santos, Eduardo PM Câmara Júnior, and Luiz FM Vieira. 2020. Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Computing Surveys (CSUR)* 53, 1 (2020), 1–36.

[47] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2022. Sound, precise, and fast abstract interpretation with tristate numbers. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 254–265.

[48] Junjie Wang, Zhiyi Zhang, Shuang Liu, Xiaoning Du, and Junjie Chen. 2023. FuzzJIT: Oracle-Enhanced Fuzzing for JavaScript Engine JIT Compiler. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1865–1882.

[49] Anatoly Yakovenko. 2018. Solana: A new architecture for a high performance blockchain v0. 8.13. *Whitepaper* (2018).

[50] Rui Yang and Marios Kogias. 2023. HEELS: A Host-Enabled eBPF-Based Load Balancing Scheme. In *Proceedings of the 1st Workshop on eBPF and Kernel Extensions*. 77–83.

[51] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294.

[52] Mingwei Zheng, Qingkai Shi, Xuwei Liu, Xiangzhe Xu, Le Yu, Congyu Liu, Guannan Wei, and Xiangyu Zhang. 2024. ParDiff: Practical Static Differential Analysis of Network Protocol Parsers. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 1208–1234.

[53] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. 2022. XRP: In-Kernel Storage Functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 375–393. https://www.usenix.org/conference/osdi22/presentation/zhong

[54] Shiyao Zhou, Muhui Jiang, Weimin Chen, Hao Zhou, Haoyu Wang, and Xiapu Luo. 2023. WADIFF: A Differential Testing Framework for WebAssembly Runtimes. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 939–950.