eBPF Misbehavior Detection: Fuzzing with a Specification-Based Oracle

Abstract

Bugs in the Linux eBPF verifier may cause it to mistakenly accept unsafe eBPF programs or reject safe ones, causing either security or usability issues. While prior works on fuzzing the eBPF verifier have been effective, their bug oracles only hint at the existence of bugs *indirectly* (e.g., when a memory error occurs in downstream execution) instead of showing the root cause, confining them to uncover a narrow range of security bugs only with no detection of usability issues.

In this paper, we propose SpecCheck, a *specification-based oracle* integrated with our fuzzer Beacon, to detect a wide range of bugs in the eBPF verifier. SpecCheck encodes eBPF instruction semantics and safety properties as a specification and turn the claim of whether a concrete eBPF program is safe into checking the satisfiability of the corresponding safety constraints, which can be reasoned automatically without abstraction. The output from the oracle will be cross-checked with the eBPF verifier for any discrepancies. Using SpecCheck, Beacon uncovered 15 bugs in the Linux eBPF verifier, including severe bugs that can cause privilege escalation or information leakage, as well as bugs that cause frustration in even experienced kernel developers.

1 Introduction

Kernel extensions are critical components of modern operating systems that allow developers to extend the kernel with custom functionality. eBPF is one such framework that allows developers to extend the Linux kernel [15]. eBPF, at its core, relies on static verification to ensure the safety of eBPF-based extentions (*i.e.*, eBPF programs). The eBPF verifier validates every eBPF program before execution, thereby preventing unsafe operations that could compromise the kernel. This role becomes even more critical with the widespread industrial deployment of eBPF-based extensions. For instance, servers at Meta each execute over 50 eBPF programs [21], underscoring the extensive reliance on these extensions in large-scale deployments. To ensure the safety of such programs, the Linux eBPF verifier statically checks each eBPF program against a set of safety properties the kernel expects (e.g., inbound memory access) using a heuristic algorithm that combines domain abstraction [22] and execution path enumeration.

Unfortunately, the eBPF verifier contains bugs that either *reject safe eBPF programs (usability issues) or accept unsafe ones (security issues)*. Specifically, rejecting safe eBPF programs imposes significant overhead on developers, who must spend time debugging correct code while struggling to understand the subtleties in the verifier [3]; Or, accepting unsafe eBPF programs impose security risks, such as privilege escalation [6] and information leakage [7].

As the bugs in the verifier can lead to critical issues, we need a deeper understanding of the current verifier design to develop effective approaches for detecting and fixing bugs. We identify four primary root causes of eBPF verifier bugs. First, abstract interpretation in the eBPF verifier is imprecise—over-approximating eBPF program states—hence, rejecting safe programs (*RC1*). Second, safety checks, implemented incrementally with new features (*e.g.*, new instructions or kernel functions) by different developers can be inconsistent, either too conservative or too relaxed (*RC2*). Furthermore, even with the required informal documentation of safety properties—actually absent, developers still make implementation mistakes (*RC3*), especially in optimization heuristics (*RC4*). Overall, RC1–3 can cause safe eBPF programs to be rejected, while RC3 and RC4 allows accepting the unsafe eBPF programs, thereby raising critical concerns.

Existing works have focused on finding a *subset* of bugs in RC3 and RC4, with less attention on RC1 and RC2. For instance, existing formal verification works specifically verify the functional correctness of a single component. Agni [40], for example, targets the range analysis component to rule out range-specific bugs (RC3). Automated testing, on the other hand, relies on specific runtime patterns, such as memory sanitizer KASAN [28, 38] or integer range discrepancies between the verifier's approximated states and eBPF program runtime states [27, 37], to detect issues in RC3 and RC4.

In this paper, we take a rather holistic approach to nip the identified root causes in their bud. We introduce the fuzzing framework Beacon, which utilizes the specification-based oracle SpecCheck that identifies eBPF verifier bugs based on the aforementioned root causes (RC1–RC4). Specifically, SpecCheck systematically specifies eBPF instruction semantics (especially the dynamic type system) and five safety properties of eBPF programs guaranteed by the eBPF VM. The specification is further encoded as pre- and post-conditions of each instruction using axiomatic notations, which is *extensible* as adding new instructions or kernel functions is simply to extend the specification with the new pre- and post-conditions. Moreover, SpecCheck leverages the specification to verify the safety of an eBPF program through SMT solvers, ensuring the *trackability* of failed conditions and the oracle as *precise* as the SMT solvers.

Beacon found 15 new bugs in total, indicating its bug-finding effectiveness. Some bugs can cause severe security consequences, such as privilege escalation to root and information leaks capable of bypassing KASLR, or usability issues

that cost eBPF developers hours to debug. 12 bugs have been acknowledged, and eight of them have already been fixed.

Summary. This paper makes the following contributions:

- **Specification of eBPF instruction semantics and safety properties**: We systematically analyzed the eBPF VM to specify its instruction semantics, particularly its type system, and the safety properties it guarantees. This provides the community with a deeper understanding of the eBPF VM, facilitating the further development of the eBPF-based extension system.
- **A specification-based oracle for holistic detection of eBPF verifier bugs**: We present the specification-based checker SpecCheck and integrate it into our fuzzer Beacon, enabling the automatic and comprehensive detection of various types of bugs in the eBPF verifier.
- **Finding severe bugs**: Beacon uncovered bugs that can lead to severe security vulnerabilities, such as privilege escalation and information leaks, as well as significant usability issues for developers.

We will open-source Beacon along with its specification and provide links to the reported new bugs to promote future research on the correctness of the eBPF verifier.

2 Background and Motivation

We introduce the basics of the eBPF virtual machine (VM) and illustrate four Linux eBPF verifier bugs uncovered by Beacon to highlight the limitations of existing works.

2.1 eBPF Virtual Machine and Verifier

Basics. The eBPF VM is a register-based architecture with a dedicated eBPF ISA [9], designed to execute kernel extensions for system tracing, security enforcement, and network processing. It has ten general-purpose 64-bit registers (`R0`-`R9`) and one read-only stack frame pointer register (`R10`), pointing to a fixed-size stack memory region private to each eBPF VM, similar to the `rbp` register on x86_64. The eBPF VM also provides other types of special-purpose memory regions to interact with the kernel and userspace (see details in the table in the supplementary material). Programs running inside the eBPF VM includes an entry function (`main`) and optionally, auxiliary functions (*pseudo functions*). Each function is a finite sequence of eBPF instructions [9].

The Linux eBPF verifier. The goal of the Linux eBPF verifier is to ensure that an untrusted eBPF program is "safe" to execute in kernel space. While we defer the discussion of safety requirements to §3.3, in a nutshell, an eBPF program cannot access arbitrary kernel data structures nor cause hangs, panics, or resource leaks.

To achieve this, the eBPF verifier implements an algorithm that preserves some safety properties. The algorithm comprises some form of abstract interpretation [22] with execution path enumeration commonly found in symbolic execution (e.g., KLEE [19]). Specifically, it first performs control flow graph validation to preclude infinite loops and recursion, and subsequently enumerates all program paths. Per each path, the verifier simulates the execution of every instruction, tracks the state change of registers and memory regions, and checks that even over-approximated execution states (*e.g.*, integers approximated with wider possible ranges) are still safe. Moreover, it "prunes" execution paths heuristically to reduces path explosion and improves verification efficiency.

Upon passing verification, eBPF programs are attached to specific kernel hooks to be executed either by the interpreter or as JIT-compiled machine code. In both cases, the safety checks done by the static verifier are not repeated at runtime.

2.2 Issues with the Linux eBPF Verifier

We now discuss the four root causes (RC) that the current verifier suffers from.

RC1: Imprecision caused by state abstraction. Even in the best-case scenario, where all contributors to the Linux eBPF verifier have a shared and complete understanding of safety properties, and the implementation is flawless, the eBPF verifier can still reject safe eBPF programs due to the inherent conservative approximation of execution states.

```
1  uint32_t array[11];
2  // x is aligned to 4 bytes
3  // x is initialized with a value load from a map
4  uint32_t x = map_value;
5  uint32_t res;
6  // Bound the range of x
7  if (x >= 0 && x < 11) {
8    res = compute_value();
9    array[x] = res; // Rejected
10 }
```

Figure 1. A bug caused by imprecise abstract interpretation and found by Beacon took a `sched-ext` developer hours to debug.

Figure 1 shows such a case discovered by Beacon. The example program contains a 44-byte local array (`array`) and a 32-bit variable `x`, which is at a 4-byte aligned address and initialized to an unknown value loaded from a map. The range of `x` is then bounded to safely index into `array` (line 7). Subsequently, `array[x]` is assigned the return value of a pseudo function `compute_value` (line 9). However, the verifier rejects it, incorrectly determining that `x` could exceed the array bounds despite the earlier bounds check.

The root cause lies in the "*intended*" abstract interpretation design of not propagating numerical range state from register to memory regions that are not 8-byte aligned. Therefore, when `x` is loaded from memory into a register `r1` and undergoes a bounds check (line 7), the verifier does not propagate the range information to the 4-byte aligned stack data. Later, `r1` and its maintained range information is overwritten when computing `res`. When `x` is subsequently reloaded into register (line 9), the verifier has no record of its previously established bounds, causing it to conservatively assume `x`

could be out of bounds and incorrectly rejecting the program. This creates a bad experience as reported by frustrated `sched-ext` [12] developers after our discovery.

```
int *ptr1 = ptr;
atomic_and(&ptr1, 1); // Success: & between 1 and ptr in &ptr1
ptr &= 1;             // Failed : & between 1 and ptr
```

Figure 2. Inconsistent safety constraints in bitwise operation

RC2: inconsistent safety rules. Ideally, participants in the eBPF ecosystem should share the same and complete understanding of safety properties. However, in reality, such systematic knowledge is nonexistent. The safety of eBPF programs is more like folklore knowledge imprecisely described in natural languages (e.g., mailing lists[4], code comments [11] and selftests [13]), and safety checks in the verifier are often implemented incrementally as the eBPF ISA expands (e.g., new instructions and kernel functions). As different developers offer individual and uncoordinated understanding of safety rules, inconsistent safety checks—checks that are more conservative or relaxed than similar counterparts without a clear justification—are not uncommon.

As shown in Figure 2, a regular bitwise `AND` and an `atomic AND` on a pointer yields inconsistent verification results, confusing eBPF application developers. Since both operations pose the same security risk (potential pointer leakage, see §3.3) with no additional safety concerns, they should be treated consistently. For privileged users who are allowed to leak pointers, both variants should be permitted to maximize programming flexibility.

```
struct bpf_iter_num it;
// memory data pointing by map_val is controlled by users.
int *map_val = ...;

bpf_iter_num_new(&it, 0, 3);
// Correct one: while (bpf_iter_num_next(&it)) {}
while (bpf_iter_num_next((struct bpf_iter_num *)map_val)) {}
bpf_iter_num_destroy(&it);
```

Figure 3. A bug that misses type checks found by Beacon.

RC3: incorrect implementation. Besides inconsistent safety notions, the implementation is rarely perfect. Developers might implement an incomplete or even wrong set of safety rules, which can cause either safe eBPF programs being rejected or unsafe eBPF programs being accepted, due to missing specifications for required safety rules.

Figure 3 illustrates a bug where the verifier fails to type check properly. The program declares a number enumeration structure with start and end fields defining a range `[0-3)`. The program then calls the kernel function `bpf_iter_num_next` in a loop to iterate the range. Internally, `bpf_iter_num_next` takes the enumeration structure, returns the current start value, and increments the start value by 1. It returns `null` when start equals end. However, due to missing type checks in the verifier, the program can pass any arbitrary memory pointer (*e.g.*, a map pointer) as the argument instead of a valid enumeration structure. For example, a user can modify data in mapped memory from userspace during runtime, potentially causing the eBPF program to loop infinitely, leading to a hang in kernel space.

RC4: erroneous optimization. Historically, a hot spot of implementation errors is the optimizations in the verifier as they increase the complexity of the verification algorithm significantly. Hence, we assign a special tag to bugs caused by optimizations and differentiate them from RC3. Specifically, to improve verification efficiency, the verifier employs path pruning via path equivalence checks. On encountering a branch starting at instruction i, the verifier checks if it has already verified this branch with a previous program state S_1. If the current state S_2 can be subsumed by S_1 the verifier skips re-verifying that branch. Essentially, the verifier only attempts to unify register values, stack data, and other states used in memory access instructions from S_2 with S_1.

Beacon discovered a bug in path pruning. The verifier implements a special behavior for programs with capability `CAP_PERFMON`, which allows leaking kernel data to userspace. When spilling an N-byte value to an uninitialized 8-byte aligned stack address, it marks the remaining $(8 - N)$ bytes as known values. This allows more states to be unified for pruning purposes. The bug arises when the verifier mistakenly applies this behavior to programs without `CAP_PERFMON`. This can be exploited to leak kernel pointers, potentially bypassing KASLR[2] and enabling more severe attacks.

```
1  int func1() {
2    uint64_t *victim_ptr;
3    // ptr @(r10-8) saves its own address
4    victim_ptr = (uint64_t *)&victim_ptr;
5    return 0;
6  }
7  int func2() {
8    uint64_t leaked_ptr; // Not initialized
9    ((char *)&leaked_ptr)[0] = 0;
10   // Bug: leaked_ptr becomes a readable 8-byte integer
11   //      but its high 7 bytes are partial victim_ptr
12   return 0;
13 }
14 SEC("socket")
15 int leak_ptr(void *ctx) { func1(); func2(); }
```

Figure 4. An erroneous optimization leads to data leakage.

Figure 4 demonstrates an exploit for this bug. The entry function `leak_ptr` (line 15) first calls `func1`, which saves a stack address in `victim_ptr` at the top of the stack frame. It then calls `func2`, which shares the same stack frame and writes a single byte to `leaked_ptr`—location that overlaps with `victim_ptr`. Due to the verifier bug, although only one byte is written, the verifier incorrectly marks the next 7 bytes as initialized integers. These 7 bytes actually contain part of the pointer value stored in `victim_ptr`. The program can then load these bytes as integers and leak them to userspace.

2.3 Improving Assurance of eBPF Verifier

Given the importance of the Linux eBPF verifier, several works try to improve its correctness, which can be broadly categorized into three themes:

Formal verification aims to prove that the eBPF verifier implementation conforms to a specific set of correctness

specifications. For example, several works [39, 40] proved the correctness of range analysis. Despite its high assurance, formal verification often struggles to scale due to the eBPF verifier's large (20K LoC) and rapidly evolving codebase. This may explain the lack of full-spectrum verification beyond range analysis.

Alternative designs with solid language-theoretic foundation also exist. For instance, Prevail [25] seeks to capture and unify ad-hoc safety rules in eBPF verifier and reimplement them with a foundational framework such as abstract interpretation. This means the alternative designs are typically more accurate with reasonable or even no performance penalties. The major drawback, however, is that it will be challenging for the alternative verifier to keep up with the rapid evolution of the eBPF verifier. For example, Prevail only supports 77 out of 455 kernel functions.

Fuzzing [23, 24, 41] as one of the most effective bug-finding approaches has been applied on eBPF verifier [27, 28, 37, 38]. While coverage-guided fuzzing can be very effective in exploring different parts of the verifier, a fuzzer still needs a bug oracle to decide when the eBPF verifier is erroneous, and the design of an eBPF-specific bug oracle has been a differentiating factor in prior works.

Bug oracles in prior works include memory sanitizer KASAN [28, 38] and a reference monitor for scalar range inconsistencies between verifier states and runtime states [27, 37]. These oracles, however, can detect only a subset of bugs caused by RC3 and RC4 as many runtime errors (e.g., information leaks or type confusion) do not necessarily lead to memory errors. These oracles also completely forgo the chance of finding issues in RC1 and RC2 as eBPF programs that fail verification will not even be executed.

2.4 Motivation and Insight

The unique characteristics of each theme drive us to consider a holistic yet practical scheme to find all bug types listed in §2.2. We present Beacon, a fuzzing-based testing framework with an oracle SpecCheck, built on an extensible set of specifications that encode constraints for eBPF programs. SpecCheck employs automated reasoning to verify eBPF programs against these specifications. If the verification result diverges from the eBPF verifier's result, SpecCheck flags it as a bug. Importantly, we do not propose SpecCheck as a replacement for the in-production eBPF verifier. Rather, SpecCheck serves as a precise and extensible testing oracle at the cost of verification speed.

We first present the design of specification-based oracle (§3), SpecCheck, which we later integrate into Beacon (§4).

3 SpecCheck: A Specification-based Oracle

Beacon aims to identify eBPF verifier bugs holistically through a specification-based oracle SpecCheck, designed with the following three goals in mind:

Goal 1: Precision. SpecCheck should accurately characterize the set of safe eBPF programs by precisely specifying program state and safety constraints. Any imprecision could lead to false alarms when comparing results with the Linux eBPF verifier, making it difficult to determine if the verifier is functioning correctly.

Goal 2: Trackability. When facing an unsafe eBPF program, SpecCheck must be able to pinpoint the root cause of safety violations. That is, SpecCheck should be able to blame the specific unsafe instruction that violated the expected eBPF safety guarantees.

Goal 3: Extensibility. The eBPF ecosystem continuously evolves with new instructions and helper functions. SpecCheck should be easily extended to model the new features, enabling thorough testing of both new, existing components, and their interplay in the eBPF verifier, without requiring significant changes to the framework.

3.1 Overview of SpecCheck

To achieve these goals, SpecCheck is built around two components: 1) The operational semantics and constraints on alignemnts and operations that the eBPF specificaiton mandates for each eBPF instruction [9] (including kernel functions). We augment this operational semantics with dynamic types and the corresponding typing rules to track the type fo values manipulated by the eBPF VM. We aim for precision by avoiding abstraction where possible (§3.2). 2) A set of safety rules for each eBPF instruction derived systematically to ensure five high-level safety properties (§3.3). These components are specified using per-instruction axiomatic notations (§3.4). This approach directly supports *extensibility*, as incorporating a new instruction only requires adding its corresponding pre- and post-conditions.

More importantly, these specifications enable SpecCheck to turn the question of *"Is a given eBPF program safe?"* into proof obligations that can be discharged to automated theorem provers, *e.g.*, satisfiability modulo theories (SMT) solvers. By encoding all dynamic checks performed in the semantics into SMT formulae, SpecCheck avoids any form of approximation, achieving a high degree of *precision*, bounded primarily by the capabilities of the underlying solver to solve the verification obligations.

Furthermore, when SpecCheck fails an eBPF program verification, the SMT solver's counterexample can be used by SpecCheck to identify the specific constraints that were violated, directly providing *trackability*.

3.2 eBPF Semantic Specification

SpecCheck encodes the semantics of eBPF instructions with their operational semantics and constraints as defined in the eBPF ISA [9]. The semantics is augmented with a dynamic type system, characterizing how types are updated by each instruction. We provide detailed definitions of term

syntax, types and dynamic typing rules in the supplementary material.

Terms. An eBPF program consists of a sequence of instructions—categorized as arithmetic, data handling, memory, and control flow operations—each of which takes immediates or registers as operands.

Types. SpecCheck defines a dynamic type system over the type domain τ, shown below.

τ ::=	`uninit`	(uninitialized data)
	| `scalar`	(1-byte to 8-byte scalars)
	| `ptrnull`(π)	(pointers of `null` or type π)
	| `ptr`(π)	(8-byte pointers of type π)

`uninit` denotes uninitialized data. After initialization, data can exist in one of three forms: a `scalar` (integers ranging from 1 to 8 bytes), a `ptr`(π), which is a guaranteed non-null 64-bit memory address pointing to a region π, or a `ptrnull`(π), representing a pointer that could either be `null` or point to a region of type π. The region type π refers to abstract memory areas such as stack, context, or packet data.

Model. SpecCheck models the eBPF VM as a state machine, in which an eBPF program comprises of functions, each containing a sequence of instructions that operate on registers and separate memory regions. SpecCheck tracks the state (*i.e.*, value V and types T) of registers and memory bytes, along with memory access permissions and other states (*e.g.*, the stack depth), forming the execution context.

For types `uninit` and `scalar`, the value V directly reflects the real value. Since pointers are not continuous, SpecCheck uses an extra state `memId` and represents pointers as <`memId`, V_{off}>, where `memId` identifies a memory region and V_{off} is the offset within it. SpecCheck does not only use π to represent each memory region as some memory types, like maps, have multiple regions.

Registers `r0`-`r9` and the stack are general-purpose, capable of storing any data type. In contrast, register `r10` and other memory regions are special-purpose and fixed to specific, unchanging types. These regions are either raw memory, where all data is of type `scalar`, or structured memory with fields of type `scalar` or `pointers`. For instance, context memory containing socket buffer data is structured with the `__sk__buff` kernel structure. Notably, since pointers are eight bytes, a pointer stored in memory indicates that all 8 bytes starting at the address must be marked with type `ptr`(π) or `ptrnull`(π).

Initial context. Initially, `r1` and `r10` represents the base address of the context and stack memory, respectively. Thus, their types `V` are set to `ptr(ctx)` and `ptr(stk)` respectively. Their values $\mathtt{T}_{off}$ are set to zero, and `memId` are set to zero and one. Other registers and memory locations are initialized as follows: Registers and stack slots are all initialized to `uninit`, while other memory bytes are initialized with their predefined types—either `scalar`, `ptr`(π), or `ptrnull`(π)—based on their intended purpose.

We now describe SpecCheck's type rules, key instruction semantics, and semantic constraints, which every eBPF program must satisfy.

Arithmetic instructions. These operations fall into two categories: single operand (*e.g.*, bitwise negation) and double operand (*e.g.*, addition). The semantics of arithmetic instructions align with those of other ISAs, except for special cases like division-by-zero or modulo-by-zero. For example, division-by-zero is allowed and returns zero.

In SpecCheck's dynamic type system, unary arithmetic instructions—bit manipulations—always update the type of `dst` register as `scalar`. Double-operand instructions behave the same, except in specific cases. When adding or subtracting a `scalar` to/from a `ptr`(π), or adding a `ptr`(π) to `scalar`, the destination register is set to `ptr`(π).

Data handling operations. Data handling includes 64-bit `mov` and `load`s. `mov` instructions copy data between registers or from an immediate to a register, while `load`s place 64-bit constants (`ptr`(π) or `scalar`) into registers. 64-bit `mov` and `load` update the destination register with the source type, while others change the destination register's type to `scalar`.

Memory operations. eBPF supports 1, 2, 4, and 8-byte general loads and stores with 4/8-byte atomic memory operations. Such operations typically require size-aligned memory accesses. SpecCheck enforces strict type rules for memory operations to prevent pointer corruption, including partial pointer loads or overwrites. Rather than tracking all memory slots, SpecCheck enforces two key constraints: stack accesses must be size-aligned, and structured memory regions access must be size-aligned and contained within a single field. This approach leverages observations that the eBPF LLVM compiler performs stack spills at size-aligned offsets and structured memory fields are all size-aligned, allowing the checker to track pointers by examining only 8-byte aligned memory blocks. Additionally, atomic operations are restricted to map memory regions since they are the only areas shared among concurrent eBPF programs, while other memory regions, like the stack, being local to single program instances, do not require atomic operations.

Based on the above semantics and constraints, we now list the type rules that SpecCheck enforces for memory operations below:

- *load*: 8-byte memory loads from the memory slots with the same type $\alpha \in \tau$ set the destination registers as α. Other loads set the destination registers to `scalar`.
- *store*: Memory regions are modeled with either mutable types (*i.e.*, stack) or immutable types (*e.g.*, context). For stack, 8-byte stores replace the slot types with the source register's type. For stores smaller than 8 bytes: if the original 8-byte slot contained a pointer type, all 8 bytes are converted to `scalar` to prevent partial pointer corruption.

Otherwise, only the targeted bytes are marked as `scalar`. For memory regions with immutable types, SpecCheck ensures slot types are compatible with the source register type—pointers can be stored as scalars, but not vice versa.

- *Atomic operations*: Maps—unstructured memory regions for `scalars` data—are the only areas where atomic operations apply. Thus, memory slot types remain unchanged and the register holding loaded data is always `scalar` type.

Control flow operations. SpecCheck executes instructions sequentially, except for jumps, function calls, and exits. Jumps transfer control within a function, either unconditionally or based on comparison results. They generally do not change data types with one exception: comparisons (== and !=) between a `ptrnull` and a `scalar` with value 0, which changes `ptrnull` to either `ptr` (non-null) or `scalar` with value 0 (`null`). Direct function calls pass up to five arguments via caller-saved registers `r1–r5`, while `r6–r9` are callee-saved. For kernel function calls, argument types are validated against the function type declarations upon entry, while other registers are set to `uninit`. On function return, `r0` holds the return value, whose value range and type are constrained by program types to ensure correct interpretation at the attachment point. For instance, the 33 program types (grouped into nine categories) each have defined return value constraints. After a function return, callee-saved registers are restored, while caller-saved registers and callee stack slots are set to `uninit`.

3.3 eBPF Safety Specification

SpecCheck derives safety properties through a top-down approach based on the CIA (Confidentiality, Integrity, and Availability) triad security model [36]. This systematic approach leads us to define three key aspects of safety:

- **Availability**: Unrestricted eBPF programs can compromise kernel availability in several ways: causing kernel crashes through memory errors, blocking kernel threads indefinitely, and depleting system resources through non-terminating execution or improper resource management. To protect kernel availability, we define three safety properties: control flow safety (SP1), memory safety (SP2), and resource safety (SP3).
- **Integrity**: The kernel integrity requires eBPF programs cannot modify unauthorized data (*i.e.*, kernel data outside of the eBPF VM and read-only eBPF registers), This requirement leads to two safety properties: memory safety (SP2) and VM integrity safety (SP4).
- **Confidentiality**: eBPF programs running in the kernel context have access to sensitive kernel data, including kernel pointers and metadata retrieved through kernel functions. This sensitive information must never be leaked to unprivileged users. If leaked, attackers can exploit these pointers to exfiltrate private data or escalate privileges by leveraging other kernel vulnerabilities [7]. To prevent such information leakage, we define data safety (SP5) as a critical security property.

Overall, we define an eBPF program is safe if the entire program satisfies the global control-flow safety and each of its instructions is safe regarding other safety properties SP2 to SP5. We now detail the control-flow safety property and each per-instruction safety property separately.

SP1: Control flow safety. eBPF programs must terminate in finite time—in terms of number of instructions executed—with an explicit `exit` instruction. Any execution of the program can have a maximum of 1M instructions for privileged users and 4096 instructions for unprivileged users. This ensures control returns properly to the kernel and prevents denial-of-service attacks and resource exhaustion.

SP2: Memory safety. SpecCheck models only discrete eBPF VM memory regions, so any unmodeled memory lies outside the VM. These unmodeled memory can be accessed either through out-of-bound VM memory access or via kernel functions. In our specification, we assume kernel functions are memory-safe. Therefore, memory errors outside the eBPF VM can only occur due to out-of-bound access on VM memory. Based on this, we enforce three key memory safety constraints: First, any dereferenced pointer must be non-null, which is verified by checking if the pointer's type is `ptr(`π`)` before memory operations. Second, memory accesses must be within bounds and have valid permissions. Memory regions in eBPF VM either have fixed or dynamic bounds. The above conditions apply to regions with fixed bounds (e.g., stack) and dynamic regions (e.g., packet) where memory bounds are determined dynamically through pointer comparisons, such as between the packet and its end pointer packet_end. In contrast, other dynamic regions (e.g., buffer and arena) depend on runtime checks. Third, to ensure temporal memory safety—preventing issues like use-after-free and double-free—we track the state of memory dynamically allocated through kernel functions. All related kernel functions are required to check the memory state before operations and update it afterward.

SP3: Resource safety. Resource safety ensures that all dynamically allocated resources are properly released before program termination. Specifically, before allowing program termination, SpecCheck checks that all dynamically allocated memory has been freed and all acquired resources (such as locks) have been released by examining the program's resource tracking state.

SP4: VM invariant/integrity. The eBPF VM enforces this invariant by making register `r10` read-only, prohibiting any instructions from writing to it as a destination register.

SP5: VM data safety. SpecCheck ensures data safety by adopting the capability model. In particular, kernel metadata is only accessible to privileged users with the `CAP_PERFMON` capability. For other capabilities, SpecCheck enforces the following data safety properties.

First, it prohibits programs from reading data with the type `uninit`, as it may contain private kernel data that was not erased. Second, it prohibits programs from storing pointers in public channels, such as map memory regions and helper calls that write to userspace memory. Still, eBPF programs can convert pointers (`ptr(`π`)` and `ptrnull(`π`)`) to scalars using arithmetic, memory, and call operations, potentially leaking these scalars to userspace. For instance, a bitwise `or` on a pointer converts it to a scalar according to the type rules described in §3.2. SpecCheck prevents such leaks by strengthening type rules that convert pointers to scalars.

- *Arithmetic instructions.* Arithmetic operations that take pointers but produce `scalars` are all disallowed.
- *Data handling operations.* Instructions that are not 64-bit `mov` and `load` and operate on pointers, are prohibited.
- *Memory operations.* The type rules in §3.2 allow converting pointers to scalars by loading or storing pointers smaller than 8 bytes. Such type transitions are prohibited to avoid kernel pointer leakage.
- *Control flow operations.* Jump instructions do not suffer from unsafe type conversions. Meanwhile, call instructions that pass arguments with pointer types to scalar parameters are already prevented in the defined type rules.

In addition, side channels can arise from two sources: (1) comparisons between pointers and scalars, and (2) speculative execution. To prevent information leakage through pointer-scalar comparisons, we prohibit such comparisons in both general jump instructions and atomic compare-and-exchange operations. But we allow `ptrnull(`π`)` to be compared with the scalar value 0 in equal/non-equal jumps to check if a pointer is `null`. For speculative execution side channels, we assume that the eBPF VM can rewrite programs with appropriate mitigations. Thus, we do not impose additional constraints on eBPF programs for this case.

Discussion of safety property completeness and soundness. The five safety properties outlined above are derived practically using a top-down approach, and Linux eBPF verifier developers confirmed that they genuinely reflect the verifier's intended guarantees. Evolving security threats and potential unknown attack vectors make it hard to define a good model of the CIA formally. As a result, we have not formally verified that our five properties ensure CIA. However, these properties remain fundamental and intuitive. Especially, they are necessary: if any were removed, a program could pass verification but compromise kernel security. *More importantly, these properties are sufficient to identify known bug types within our bug-finding scope.*

3.4 Specification Encoding

Dafny, in particular, offers a unique feature known as the ghost method, which allows the expression of pre- and post-conditions for unimplemented methods purely for verification purposes. This enables us to specify the behavior of each instruction modularly. It also tracks violated constraints when these conditions fail, enhancing the debugging and refinement process.

```
1  ghost method {:axiom} neg32(dst: RegState)
2    // Pre-conditions: safety properties
3    requires dst.regNo != R10                              // SP4
4    requires !allow_ptr_leak ==> dst.regType == SCALAR  // SP5
5    modifies dst
6    // Post-conditions: instruction semantics
7    ensures dst.regType == SCALAR                  // Type rule
8    ensures if old(dst.regType) == SCALAR
9            then dst.regVal == !old(dst.regVal) // Value update
10           else unknownBv32(dst.regVal)
```

Figure 5. The encoded specification of instruction `neg32`, which flips a 32-bit value in the register `dst`.

Concretely, the specification is defined as a Dafny class, with registers and memory regions as global variables. Registers are modeled as structures with fields for metadata (e.g., `regNo`, `regVal`, `regType`), while memory regions are arrays of 8-bit vectors. Instructions are encoded as ghost methods, including only safety properties and instruction semantics as pre- and post-conditions for verification purposes.

We encode each instruction's specification based on its operational semantics from the ISA [9], the semantic constraints and type rules in §3.2, and safety properties in §3.3. For kernel functions whose semantics are not detailed in the ISA, we extract both operational semantics (e.g., return values) and semantic constraints (e.g., `spin_lock` requires no already held locks) from their kernel implementations. To prevent encoding errors in safe properties (e.g., missing safety properties), we map each safety property with its corresponding semantic component: 1) SP2 (memory safety) with memory read/write/allocation/release semantics, 2) SP3 (resource safety) with program termination semantics, 3) SP4 (VM integrity) with register write semantics, and 4) SP5 (data safety) for all instruction semantics. Using the mapping, we can identify the necessary safety properties of each instruction by matching its semantic with the safety properties' semantic component.

Figure 5 illustrates the encoding of the 32-bit bitwise negation instruction `neg32`, whose semantics involves register write, thus having the VM invariant **SP4** and the data safety property **SP5** as preconditions. SP4 requires the modified register (`dst`) must not be the read-only register `R10`. SP5 enforces this instruction to compute on `scalar` data if the privilege is missing, avoid leaking pointers or uninitialized values. If the safety properties hold, according to the type rule, the type of destination register transits to `scalar`. Further, the instruction semantic regarding data value is encoded with the negation operation on the input value (*i.e.*, `!old(dst.regVal)`). Specifically, SpecCheck models pointers as a combination of a memory region and an offset, rather than as an absolute pointer value. Thus, we cannot negate the offset (`dst.regVal`) as the computed result and instead assign an uknown 32-bit (`unknownBv32()`) value to it.

Algorithm 1: The embedding process in SpecCheck

```
1  function Embed(insns, limit)
2      insnStack, idx, output ← [], 0, ""
3      while idx < limit do
4          insn = insn[idx]
5          output+ = ISA2Method(insn)
6          if UncondJmp(insn) then
7              idx = JmpTarget(insn)
8          else if CondJmp(insn) then
9              PushInsn(insnStack, JmpTarget(insn))
10             idx = idx + 1
11         else if PseudoCall(insn) then
               ▷ Inline pseudo calls
12             PushInsn(insnStack, nextInsn)
13             idx = FunEntry(insn)
14         else if Exit(insn) then
15             if Empty(insnStack) then
16                 break ▷ Break at the last instruction
17             else
18                 idx = PopInsn(insnStack)
19         else
20             idx = idx + 1
21     return output
```

4 Marrying Specification with Fuzzing

With the encoded specification—per-instruction semantics and safety properties—defined in Dafny (§3), the next step is to turn SpecCheck into a bug oracle that can be integrated with fuzzers, which is the focus of this section. In particular, we show how to verify whether a concrete eBPF program is safe or not via shallow embedding [26] and present the complete fuzzing workflow in §4.2.

4.1 Shallow Embedding

Shallow embedding maps components of a source language S to corresponding elements of a target language D. This paper maps eBPF instructions to Dafny. Algorithm 1 outlines the embedding procedure. The algorithm performs a depth-first traversal of the program's control flow graph, mapping each instruction to its corresponding method in the specification (line 5). During traversal, the algorithm handles control flow instructions specially: (1) For unconditional jumps, it jumps to the target instruction. For conditional jumps, it explores both the fallthrough path and the jump target path; (2) For pseudo functions calls, it inlines them and jumps to the call entry and saves the location of the next instruction; (3) On the `exit` instruction, it restores any saved locations to explore remaining paths. The embedding procedure itself performs control flow validation, complementing the checks done by the Dafny verifier. For loops, the algorithm uses bounded unrolling to ensure termination.

Figure 6 illustrates the shallow embedding approach, where eBPF assembly code is mapped line-by-line to the corresponding Dafny method invocations. For example, the instruction (line 2) loads a 64-bit value from the start of the context memory region (pointed to by register `r1`) into register `r2` and is embedded as a call to Dafny's `load64` method. Similarly, conditional jumps are embedded as `if` combined with appropriate comparison method calls in Dafny.

```
1  //
2  r2 = *(u64 *)(r1 +0)
3  r3 = 2
4  r2 %= r3
5  if r2 == 0 goto L1
6    ...
7    call map_lookup_elem
8  L1:
9  exit
```

(a) eBPF assembly code.

```
1  var s := new Spec();
2  s.load64(s.r2, s.r1, 0);
3  s.mov64_reg(s.r3, 2);
4  s.mod64_reg(s.r2, s.r3);
5  if (!s.jeq64_imm(s.r2, 0)) {
6    ...
7    s.map_lookup_elem();
8    s.exit();
9  } else {s.exit();}
```

(b) Dafny code embedding eBPF.

Figure 6. The Dafny program shown in (b) is the embedding of the eBPF program in (a), which is in the form of eBPF assembly code. Each instruction in (a) is line-to-line mapped to (b).

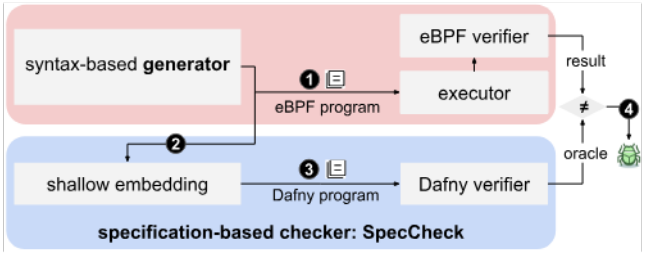

Figure 7. Beacon overview. The generator produces syntax-valid eBPF programs as test cases. Further, the eBPF verifier takes them to verify their safety (❶). Simultaneously, our checker (SpecCheck) embeds these test cases into Dafny programs based on our specification (❷) and check their safety (❸) through the Dafny verifier, which serves as an oracle. Finally, a difference between the eBPF verifier result and the SpecCheck oracle indicates a bug (❹).

4.2 Overall Fuzzing Workflow

In this subsection, we present the integration of SpecCheck as a testing oracle within the Beacon fuzzing framework. Figure 7 shows the workflow of Beacon.

Initially, the syntax-based `generator` produces test cases following the eBPF ISA. Further, the `executor` runs test cases through the eBPF verifier, which outputs the verification result, either accepting or rejecting programs (step ❶). Simultaneously, Beacon uses SpecCheck to independently validate the safety of the test cases against specifications encoded in Dafny. This involves embedding eBPF programs in Dafny (step ❷) and running the Dafny verifier to ensure they comply with the specifications (step ❸). Any discrepancy between the eBPF verifier results of the eBPF verifier and SpecCheck indicates a bug in the eBPF verifier (step ❹). Specifically, discrepancy arises when (1) programs deemed safe by the eBPF verifier are flagged unsafe by SpecCheck, (2) the reverse occurs, or (3) both reject the program but cite different safety violations. In the third case, while the eBPF verifier's final verdict on the program's safety is correct, its subprograms may reveal incorrect verification results, belonging to one of the first two cases.

The Dafny verifier relies on SMT solvers to determine if the pre- and post-conditions hold. While SMT solvers are precise, they can be computationally expensive and may even timeout or become undecidable when faced with complex or numerous constraints [18]. We use three key insights to address the scalability challenge of SMT solvers.

Small test cases. Most eBPF verifier bugs can be triggered by small, simple programs rather than large, complex ones. Our evaluation in §6.1 shows that the proof-of-concept (PoC) programs for existing eBPF verifier bugs are relatively small, mostly containing only 1–30 instructions. This insight allows us to configure our test case generator to focus on generating small programs, which reduces SMT-solving time while maintaining effectiveness at finding bugs.

Incremental state sampling for fast verification. When the eBPF verifier rejects a program, it identifies the unsafe / culprit instruction. To check if the rejection is correct, we only need to verify the safety of that instruction rather than checking the entire program. However, SpecCheck still needs information about the VM state immediately before that instruction executes. We instrument the eBPF verifier to generate VM states incrementally at configurable intervals during its verification process. We then use the sampled state closest to the culprit instruction as the initial state for SpecCheck. This sampling approach significantly improves performance, as we demonstrate in §6.3.

Parallel verification for high throughput. To further improve the fuzzing throughput on top of the above two solutions, Beacon runs the checker asynchronously from the fuzzing loop and simultaneously on multiple test cases.

Although the insights help, 0.2% of tests still time out (see §6.1). However, it is not a concern in fuzzing. If a test that might trigger a bug times out, we simply skip it, as another solvable test will likely reveal the same bug.

5 Implementation

We integrate Beacon into Syzkaller [23] to leverage its existing fuzzing components (e.g., executor). The implementation of each component in Beacon is detailed below.

SpecCheck. The specification is encoded in Dafny [10] with 2000 lines of code, and the embedding procedure is implemented in 600 lines of C++.

eBPF program generator. The generator, written in 2,100 lines of C++ code, produces a random control graph with vertices as basic blocks, iterates over each to generate syntax-valid instructions adhering to basic safety rules (e.g., avoiding r10 as a destination register), and records the program state coarsely, including initialized registers and their types.

eBPF verifier state sampling. We patch the Linux kernel to sample verifier state every N instructions, capturing registers, stack slots, and `spin_lock` statuses, without altering verifier logic. The state is shared to userspace via `debugfs`.

6 Evaluation

We evaluate Beacon on the Linux eBPF verifier to answer the following research questions.

- **Q1.** How effective and accurate is Beacon in finding eBPF verifier bugs? (§6.1)
- **Q2.** How does SpecCheck outperform other oracles? (§6.2)
- **Q3.** What is the testing performance of Beacon? (§6.3)
- **Q4.** What is the required effort to extend the specification for new instructions? (§6.4)

Experiment setup. We evaluate Beacon on Ubuntu 22.04.4 LTS with a 224-core Intel(R) Xeon(R) Platinum 8276L processor and 754G memory. We use Dafny CLI V4.6.0 and Z3 V4.12.1 [8] to verify embedded eBPF programs in Dafny.

6.1 Bug-Hunting Result

After running Beacon intermittently for three months, it uncovered 15 bugs: three cases where unsafe eBPF programs are accepted, nine cases where safe eBPF programs are rejected, one case where programs misusing atomic instructions on local memory are accepted (bug #9, see analysis in §6.5), listed in Table 1, as well as one memory bug and one undefined behavior identified through `KASAN` and `UBSAN` in the verifier itself, though the latter two are beyond the focus of the work. These bugs, found despite extensive testing by maintainers and prior research, underscore the incapability of existing oracles. We reported all 15 bugs, of which 12 are acknowledged. The remaining three (bugs #10, #11, and #13) are usability issues rooted in the core design limitations of the verifier. These are typically considered low priority by developers compared to security issues and eBPF ISA extensions and are therefore often overlooked. We emphasize that Beacon not only identifies misimplementation bugs, but also highlights existing design limitations, offering insights that improve the verifier in the future. Eight bugs were fixed quickly. The rest are pending resolution for two reasons: (1) some require non-trivial code refactoring or new algorithms, which demands significant development time, and (2) others represent usability issues rather than security vulnerabilities, which maintainers have prioritized lower in their roadmap. We are actively collaborating with eBPF maintainers to develop and submit patches for the outstanding issues.

Further, we summarize the characteristics of bugs below:

Critical consequences. New bugs can lead to severe security attacks and usability issues. For example, bug #1 and #2 can be exploited by users only with `CAP_BPF` to escalate into root and leak kernel pointers, breaking KASLR. Moreover, bug #6 took a `sched-ext` developer hours to debug it.

Diverse culprit instructions. New bugs lie in the incorrect verification of all instruction categories, *e.g.*, arithmetic (bug #7), data handling (bug #1), memory (bug #5), and control flow operations (bug #2), demonstrating SpecCheck is general enough for finding bugs throughout the entire verifier.

RC	#	Instruction	Status	Description and Consequence
RC4	1	`mov`	Fixed	Fails to track non-r10 precision on stack, leading to privilege escalation.
RC3	2	`kfunc call`	Fixed	Miss argument type checks, leading to DoS.
RC4	3	`store`	Fixed	Incorrectly mark stack slot type, leading to ASLR bypass.
RC3	4	`atomic*`	Fixed	Miss propogating precisions to stack slots used in atomic instructions.
RC3	5	`atomic_xchg`	Acked	Verifier misidentifies scalar type, failing stack pointer validation.
RC1	6	`store`	Acked	Not propagate scalar range from registers to stack
RC3	7	`be32`	Fixed	Incorrect precision back-propagation
RC3	8	`store`	Fixed	Mis-reject a 32-bit store to ovewrite a spilled 64-bit scalar on the stack.
RC2	9	`atomic*`	Acked	Allow atomic instructions operating on local memory regions.
RC2	10	`arith operations`	Reported	Inconsistent constraints on instructions converting pointer to scalars
RC1	11	`jumps`	Reported	Coarse-grained pointer comparision
RC1	12	`memory operations`	Acked	Imprecise stack data tracking
RC1	13	`arith operations`	Reported	Inaccurate tracking of arithmetic instruction result

Table 1. The list of bugs detected by Beacon. kfunc call: kernel function call

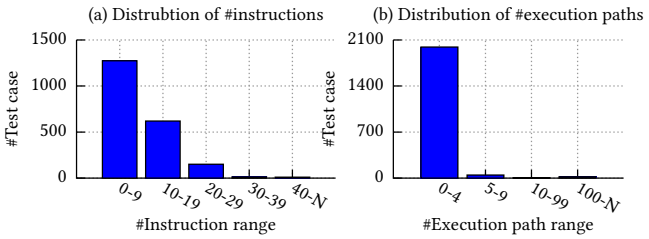

Figure 8. The size of test cases (self-tests and bug PoCs) measured by instruction count and execution paths, indicating that small-sized test cases are sufficient for finding bugs.

Small-sized test cases. Test cases revealing new bugs are as small as two instructions, as demonstrated by bugs #3, #5, and #7. To validate that eBPF verifier bugs can be detected with small test cases, we analyzed the size of various test cases, including eBPF self-tests and collected bug proofs-of-concept (PoCs). As shown in Figure 8, most test cases contain 1-30 instructions (a) and 1-9 execution paths (b), with only 23 and 28 test cases exceeding 30 instructions and ten execution paths, respectively. These findings support the insight in §4 that small test cases are sufficient to uncover most bugs.

Accuracy of the specification-based oracle. During the entire intermittent running, SpecCheck reported no false alarms but encountered timeouts (0.2% of tests) due to test complexity. These timeouts occur when the Dafny verifier encounters undecidable constraints or exceeds 900 seconds.

6.2 Comparison with the State-of-the-art

To demonstrate the effectiveness of Beacon, we compare it with existing fuzzers [27, 28, 37, 38] from two perspectives.

First, we evaluate if Beacon can detect bugs identified by existing fuzzers. This paper focuses on the oracle SpecCheck—instead of a test generator—which can be integrated into other fuzzers to leverage their test case generator with engineering effort. Therefore, we directly evaluate SpecCheck using PoCs from a collected bug dataset. The dataset includes 14 verifier bugs reported by existing fuzzers (listed in the supplementary materials): 10 from SEV, 1 from BVF, 2 from Buzzer, and 1 from BRF. We excluded bugs outside Beacon's scope, such as those in the JIT compiler or kernel function implementations. While SEV claimed 14 bugs (comprising 12 incorrectly accepted unsafe programs and 3 incorrectly rejected safe programs), we could only identify 11 of these through mailing lists, as individual bugs were poorly documented and difficult to trace. It's worth noting that SEV's oracle has an inherent limitation: it cannot detect incorrectly rejected safe programs as its oracle relies on the runtime states but rejected programs even have no chance to run. SEV identified incorrectly rejected safe programs in our dataset either manually or through the verifier's self-assertions. We also excluded two additional "bugs" from SEV and BVF that only produced misleading error messages without affecting verification outcomes. SpecCheck detected all 14 bugs in the dataset, demonstrating its comprehensive capability in identifying existing verifier bugs.

We also evaluated if existing fuzzers can detect the bugs found by Beacon. Among the four fuzzers considered, only BRF [28] and Buzzer [27] are open-source, allowing for direct empirical comparison, while SEV [37] and BVF [38] were evaluated through theoretical analysis. To ensure a fair comparison that eliminates randomness in test generation, we provided the exact bug PoCs directly to Buzzer and BRF. Despite this advantage, neither fuzzer successfully detected any of the bugs found by Beacon. Even for bug #1, which can potentially cause runtime memory errors, both fuzzers failed to find the correct map data to trigger the faulty instruction in the provided PoC. Our theoretical analysis on SEV and BVF indicates that existing fuzzers may detect bug #1, linked to runtime memory errors. The remaining bugs do not manifest as runtime errors, which clearly demonstrates the necessity of our specification-based oracle. These results conclusively show that SpecCheck can identify bugs that are beyond the detection capabilities of existing oracles.

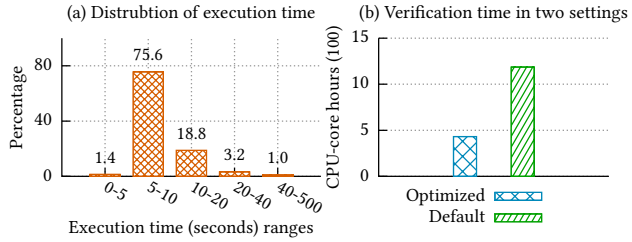

Figure 9. Fuzzing performance. (a) shows the entire execution time distribution of test cases during fuzzing, while (b) depicts the Dafny verification time without and with sampled verifier states.

6.3 Fuzzing Performance

We ran the fuzzer for 40 hours, generating eBPF programs with 5-30 instructions based on test case size analysis in §6.1, with a Dafny verification timeout of 900 seconds and state sampling every three instructions after six instructions.

Fuzzing throughput. State sampling and asynchronous checking during evaluation achieve a fuzzing speed of 23-25 tests per second by utilizing all 224 cores. The checking time of each test case in SpecCheck, including state sampling, embedding, and Dafny verification, takes 1.5 to 489 seconds, averaging 10 seconds. As shown in Figure 9 (a), 95.8% of test cases complete within 5-20 seconds, with few exceeding 20 seconds. Stage-wise analysis reveals Dafny verification consumes 99% of the time, while state sampling and embedding account for just 0.03% and 0.0002%.

Performance improvement from state sampling. We reran the fuzzer for 40 hours with the same setting in the above evaluation, except verifying each test case twice: once with state sampling (*optimized setting*) and once by verifying the complete test case without state sampling (*default setting*). Beacon generated 143,733 test cases, with 51,254 using state sampling. On average, state sampling saved 53 seconds per test case, with a maximum saving of 12 minutes. However, state sampling introduces longer checking time on 5.0% (2,574) test cases because SpecCheck identifies less complex violated constraints in another culprit instruction in the default setting. Despite this, state sampling reduces overall resource usage, saving 754 CPU-core hours [1], as illustrated in Figure 9 (b). Moreover, state sampling does not introduce bug misses in the evaluation.

Code coverage. Beacon aims to provide a comprehensive testing oracle rather than a dedicated test generator like BVF and BRF, with test generation improvements planned for future work. Nevertheless, our system achieves 32% branch coverage (3,432 branches), which exceeds BRF's 29% (3,124 branches)—BRF being the only publicly available system with comparable metrics. This coverage level has proven sufficient for discovering numerous bugs. It is worth noting that our coverage measurement uses XOR operations on basic block IDs, a method that may introduce hash collisions and potentially underestimate the actual coverage achieved.

[1] A CPU-core hour measures the usage of one CPU core for an hour [14].

6.4 Specification Extensibility and Complexity

To extend the specification with new instructions or kernel functions, users are expected to write more ghost methods in Dafny, embedding safety rules as pre-conditions and operational semantics as post-conditions. While it is hard to measure the actual manual effort scientifically, we estimate it by counting the pre- and post-conditions required for each instruction in the current specification.

On average, each instruction or kernel function has 3.6 pre-conditions (max 13) and 3.7 post-conditions (max 6). Arithmetic and data-handling instructions have fewer pre-conditions (2.9) but more post-conditions (4.7), because they generally only require operand type checks but involve special pointer semantics due to representing pointers as memory regions and offsets. Conversely, memory operations and kernel functions generally have more pre-conditions (4.7) and fewer post-conditions (2.3). The more pre-conditions stem from fine-grained memory access control and program state checks, while fewer post-conditions arise because general-purpose memory regions, excluding stack memory, have simpler type rules. Additionally, kernel functions typically only specify return values as post-conditions, except in complex cases, such as "spin_lock", which often involve changes to program states (e.g., lock state).

6.5 Case Study

We present three additional bugs found by Beacon to further illustrate their characteristics.

```
1                        r1 = r10
2                        if cond
3                   (b1) /     \ (b2)
4                       /       \
5     *(u64 *)(r1 -120) = 0   *(u64 *)(r1 -120) = r2
6                       \       /
7                    r2 = *(u64 *)(r1 -120)
8           // r3 points to an eight-byte memory region
9                        r3 += r2
10                    *(u64*)(r3 + 0) = evil_data
11  // Memory write from (b2) can overwrite any kernel addresses.
```

Figure 10. An erroneous optimization in Linux eBPF verifier prunes instructions after line 7 in branch *b2*, causing out-of-bound access.

Bug #1: Out-of-bound access leading to privilege escalation. The bug in Figure 10 is a mis-optimization bug originating from RC4. It overlooks `r2` and stack slot `r1-120` as critical states, skips their comparison at the convergence point (line 7) when verifying branch (b2), and incorrectly deems the program states from branches (b1) and (b2) equal, leading to skipped verification of instruction beyond line 7 in branch (b2). This allows branch (b2) to access any kernel address via `r2`, enabling privilege escalation with only the `CAP_BPF` capability. The vulnerability has been fixed and merged to the stable Linux version.

Bug #8: Mis-rejecting 32-bit variable initialization. In Figure 11, the eBPF verifier incorrectly rejects the initialization of a 32-bit integer array, citing *"attempt to corrupt spilled pointer on stack"*, if the program has no data leakage

```
uint32_t array[4];
array[0] = 1;  // fp-8
array[1] = 2;  // fp-4
// Reject: attempt to corrupt spilled pointer on stack
```

Figure 11. Initialization of a 32-bit integer array is mistakenly rejected by the Linux eBPF verifier. `fp` represents the stack frame pointer. `fp-8` means the stack slot at `fp-8`.

privilege. This error arises because the verifier assumes an 8-byte aligned stack slot contains a pointer without verifying its type. Specifically, after initializing `array[0]` at `fp-8` with the integer constant 1, the slot type is scalar, not pointer. When initializing `array[1]` at `fp-4`, the verifier checks the 8-byte region ([`fp-8`, `fp-1`]) for pointers to prevent partial overwrites and potential pointer leakage. However, it does not check the type of the region and erroneously treats it as a pointer, leading to the rejection. Without SpecCheck, no existing oracles can detect such bugs.

Bug #5 and #9. Atomic operations are intended for concurrent access to shared memory. However, the verifier allows unnecessary atomic instructions on private memory (*e.g.*, stack), as in the acknowledged bug #9. Although not directly causing usability or security issues, this increases implementation complexity and contributes to bug #5. The issue arises with `atomic_xchg(r1, r1)` in privilege mode, where `r1` is a stack pointer. This instruction swaps the value at the address pointed to by `r1` with the value in `r1`, involving a load followed by a store. The verifier checks the load and then the store, but immediately marks r1 as a scalar after the load, leading to a failure in the store check since r1 is no longer a pointer. Finally, this safe instruction is mistakenly rejected.

7 Discussion

Semantics fidelity. To ensure that SpecCheck is faithfully modeling eBPF, we tested that: 1) SpecCheck conforms to all selftests of the Linux eBPF verifier, and 2) all discrepancies between SpecCheck and the Linux eBPF verifier found during fuzzing can be attributed.

Proving properties about our specification. SpecCheck derived safety properties based on three fundamental security principles: availability, integrity, and confidentiality (CIA). However, mainly for a lack of formal definitions of the three security principles, we could not formally verify that SpecCheck guarantees the three security principles. Future work could formally define properties corresponding to the CIA in a simplified attacker model and then attempt to prove that our 5 safety properties are sufficient to guarantee the CIA. We have preliminarily started exploring formalization of integrity and confidentiality. Once such definitions and proofs are completed, SpecCheck would enable another purpose: it could help developers prevent errors when adding new instructions or making changes to the eBPF verifier by enabling them to verify that the updated specifications are still ensuring the formally defined CIA properties. Further, this could be checked in the eBPF development CI/CD pipeline.

Beyond testing oracle. Beyond acting as a testing oracle, the specification in SpecCheck can be further leveraged to enable the automatic generation of formal proofs in proof-carrying code (PCC) [32, 33] in eBPF-based kernel extensions. PCC requires code producers (*e.g.*, extension developers) to provide formal proof of code safety, allowing consumers (*e.g.*, OS kernels) to verify compliance with safety properties, offering developers more flexibility in proving code safety.

Handling loops in SpecCheck. SpecCheck verifies eBPF programs with loops using bounded loop unrolling, which is sufficient as a testing oracle as most of the generated eBPF programs are small as shown in the bugs found in §6.1. An alternative is automated loop invariant inference or generating eBPF programs based on pre-defined loop invariants. We leave them to future work.

8 Related Work

We discussed related work on eBPF verifier testing in §2.3 and address the remaining related work below.

eBPF verification. Formal verification has improved the correctness of the eBPF JIT compiler [34] and range-tracking in the verifier [40]. Verifying the eBPF verifier is more complex due to its 20,000 lines of code and intricate logic, compared to JIT's 2,440 lines for one architecture [5]. While we do not verify the verifier, our specification, defining instruction semantics and safety properties, can be leveraged to verify eBPF verifier further.

Specification-based testing oracle. The challenge of specification-based oracles involves defining the specification and encoding it as an executable oracle. Some studies offer frameworks [16, 20] and languages [1] that support these stages with assertions, while others focus on encoding using informal specifications like POSIX or TCP/IP RFCs—either generating formal specifications automatically (e.g., rule-based methods [31]) or manually encoding them, as seen in Netsem [17] and SibyiFS [29, 30, 35]. In eBPF, the lack of formal documentation forces us to create a systematic specification, a difficult yet key contribution. Moreover, our per-instruction specification with pre- and post-conditions introduces a novel, extensible approach that integrates seamlessly with shallow embedding for fuzzing-based testing oracles.

9 Conclusion

Specification-based oracle offers a holistic approach to detecting correctness bugs. In the Linux eBPF verifier case, Beacon finds high-impact bugs across diverse root causes, some of which can never be found with bug oracles in prior fuzzing campaigns as these oracles only indirectly hint the existence of a bug without revealing its root cause.

References

[1] Test Oracles. http://ix.cs.uoregon.edu/~michal/pubs/oracles.pdf, 2001.

[2] Kernel address space layout randomization. https://lwn.net/Articles/569635/, 2013.

[3] Avoid verifier failure for 32bit pointer arithmetic. https://lore.kernel.org/bpf/20200618234631.3321118-1-yhs@fb.com/, 2020.

[4] A discussion mail in the eBPF mailing list. https://lore.kernel.org/bpf/20230502005218.3627530-1-drosen@google.com/, 2023.

[5] Complexity of the BPF Verifier. https://pchaigno.github.io/ebpf/2019/07/02/bpf-verifier-complexity.html, 2023.

[6] CVE-2023-2163 of the eBPF verifier that leads to privilege escalation. https://nvd.nist.gov/vuln/detail/cve-2023-2163, 2023.

[7] Leak kernel points by exploiting spectre in eBPF programs. https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf.git/commit/?id=e4f4db47794c, 2023.

[8] The Z3 Theorem Prover . https://github.com/Z3Prover/z3, 2024.

[9] BPF Instruction Set Architecture (ISA). https://www.rfc-editor.org/rfc/rfc9669.html, 2024.

[10] Dafny Reference Manual. https://dafny.org/dafny/DafnyRef/DafnyRef, 2024.

[11] eBPF verifier source code. https://github.com/torvalds/linux/blob/master/kernel/bpf/verifier.c, 2024.

[12] Sched_ext Schedulers and Tools. https://github.com/sched-ext/scx, 2024.

[13] The eBPF selftest set. https://github.com/torvalds/linux/tree/master/tools/testing/selftests/bpf, 2024.

[14] What are Core-Hours? How are they estimated? https://support.onscale.com/hc/en-us/articles/360013402431-What-are-Core-Hours-How-are-they-estimated, 2024.

[15] Linux eBPF. https://ebpf.io/, 2025.

[16] Shadi G Alawneh and Dennis K Peters. Specification-based test oracles with junit. In *CCECE 2010*, pages 1–7. IEEE, 2010.

[17] Steve Bishop, Matthew Fairbairn, Hannes Mehnert, Michael Norrish, Tom Ridge, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with logic: Rigorous test-oracle specification and validation for tcp/ip and the sockets api. *Journal of the ACM (JACM)*, 66(1):1–77, 2018.

[18] Cristina Borralleras, Daniel Larraz, Enric Rodríguez-Carbonell, Albert Oliveras, and Albert Rubio. Incomplete smt techniques for solving non-linear formulas over the integers. *ACM Transactions on Computational Logic (TOCL)*, 20(4):1–36, 2019.

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

[20] David Coppit and Jennifer M Haddox-Schatz. On the use of specification-based assertions as test oracles. In *29th Annual IEEE/NASA Software Engineering Workshop*, pages 305–314. IEEE, 2005.

[21] Jonathan Corbet. BPF at Facebook (and beyond). https://lwn.net/Articles/801871/, 2019.

[22] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977.

[23] David Drysdale. Coverage-guided kernel fuzzing with syzkaller, 2016.

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

[25] 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*, pages 1069–1084, 2019.

[26] Jeremy Gibbons and Nicolas Wu. Folding domain-specific languages: deep and shallow embeddings (functional pearl). In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 339–347, 2014.

[27] Google. Buzzer - an ebpf fuzzer toolchain. https://github.com/google/buzzer.

[28] Hsin-Wei Hung and Ardalan Amiri Sani. Brf: Fuzzing the ebpf runtime. *Proceedings of the ACM on Software Engineering*, 1(FSE):1152–1171, 2024.

[29] 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)*, Ontario, Canada, October 2019.

[30] Tao Lyu, Liyi Zhang, Zhiyao Feng, Yueyang Pan, Yujie Ren, Meng Xu, Mathias Payer, and Sanidhya Kashyap. Monarch: A fuzzing framework for distributed file systems. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 529–543, 2024.

[31] Manish Motwani and Yuriy Brun. Automatically generating precise oracles from structured natural language specifications. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 188–199. IEEE, 2019.

[32] George C Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, 1997.

[33] George C Necula and Peter Lee. Safe kernel extensions without run-time checking. In *OSDI*, volume 96, pages 229–243, 1996.

[34] 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 *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 41–61, 2020.

[35] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. Sibylfs: formal specification and oracle-based testing for posix and real-world file systems. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, October 2015.

[36] Spyridon Samonas and David Coss. The cia strikes back: Redefining confidentiality, integrity and availability in security. *Journal of Information System Security*, 10(3), 2014.

[37] Hao Sun and Zhendong Su. Validating the ebpf verifier via state embedding. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 615–628, 2024.

[38] Hao Sun, Yiru Xu, Jianzhong Liu, Yuheng Shen, Nan Guan, and Yu Jiang. Finding correctness bugs in ebpf verifier with structured and sanitized program. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 689–703, 2024.

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

[40] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. Verifying the verifier: ebpf range analysis verification. In *International Conference on Computer Aided Verification*, pages 226–251. Springer, 2023.

[41] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)*, 2022.