Validating the eBPF Verifier via State Embedding

Hao Sun and Zhendong Su, *ETH Zurich*

https://www.usenix.org/conference/osdi24/presentation/sun-hao

This paper is included in the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-40-3

Validating the eBPF Verifier via State Embedding

Hao Sun
ETH Zurich

Zhendong Su
ETH Zurich

Abstract

This paper introduces *state embedding*, a novel and highly effective technique for validating the correctness of the eBPF verifier, a critical component for Linux kernel security. To check whether a program is safe to execute, the verifier must track over-approximated program states along each potential control-flow path; any concrete state not contained in the tracked approximation may invalidate the verifier's conclusion. Our key insight is that one can effectively detect logic bugs in the verifier by embedding a program with certain approximation-correctness checks expected to be validated by the verifier. Indeed, for a program deemed safe by the verifier, our approach embeds concrete states via eBPF program constructs as correctness checks. By construction, the resulting state-embedded program allows the verifier to validate whether the embedded concrete states are correctly approximated by itself; any validation failure therefore reveals a logic bug in the verifier. We realize *state embedding* as a practical tool and apply it to test the eBPF verifier. Our evaluation results highlight its effectiveness. Despite the extensive scrutiny and testing undertaken on the eBPF verifier, our approach, within one month, uncovered 15 previously unknown logic bugs, 10 of which have already been fixed. Many of the detected bugs are severe, *e.g.,* two are exploitable and can lead to local privilege escalation.

1 Introduction

The Extended Berkeley Package Filter (eBPF) [32, 37] allows untrusted user space extensions to be executed in kernel space. This mechanism has been broadly adopted by modern operating system kernels to flexibly implement various specialized tasks, including filtering [43], profiling [28], and security monitoring [15], among others [23, 54]. To ensure the safety of the untrusted extensions, a static checker [8] (verifier) is utilized to rigorously validate their integrity. In this work, our primary focus is on the eBPF verifier in the Linux kernel, which is mature and has successfully been applied in various contexts. The eBPF verifier employs abstract interpretation [20], a process where it traverses the program and gathers approximations across different abstract domains to identify potentially invalid behaviors. Due to its intricate checking mechanism, the verifier has evolved into one of the most complex components within the eBPF subsystem.

The correctness of the eBPF verifier is of utmost significance. The eBPF subsystem provides extensibility by granting restricted kernel space code execution capability to user space, which is enforced by the verifier. These restrictions are crucial as they limit memory access and control flow in programs, thereby preventing the kernel from being impacted by potentially harmful extensions. However, logic bugs in the verifier can compromise these restrictions, leading to unsafe programs being loaded. Indeed, the verifier's vulnerabilities are attractive to attackers as these bugs have a higher likelihood of being exploited to inject malicious programs into the kernel [1–3]. We will demonstrate, in Section 2.2, the exploitation of one such bug we found, a simple incorrect type cast in the verifier, to achieve local privilege escalation. Therefore, detecting and rectifying logic bugs in the eBPF verifier is critical to the overall kernel security.

Given its importance, existing work applies formal verification to several components of the verifier. For example, Agni [45] generates verification conditions for the range analysis of the verifier, and other work [44, 52] aims to verify the `tnum` domain [35]. These efforts have provided strong guarantees for the correctly verified components. Nevertheless, given its complexity, obtaining specifications, either manually or automatically, is intrinsically challenging even for a portion of the verifier [18]. Consequently, these checked specifications may be incomplete [9] or diverge from the implementation [4], *e.g.,* we still uncovered logic bugs in the verified range analysis. Moreover, these components are relatively small, and the verifier is constantly evolving with new algorithmic enhancements and features. Previous work has also applied automatic testing on eBPF [25, 26, 42]. For instance, Syzkaller [47] has been incorporated into the eBPF upstream and has identified many memory errors in the eBPF system call. Yet, they encounter challenges in detecting logic bugs due to the lack of effective test oracles [24], namely methods to automatically determine whether a program should be accepted or rejected by the verifier.

Observation. In essence, the verifier checks eBPF programs by tracking states at different locations along each possible execution path within its abstract domains, *i.e.,* the verifier state (approximation). The checking procedure on each execution path can be modeled as verifier state transitions:

$$A_0 \rightarrow A_1 \rightarrow \ldots \rightarrow A_{n-1} \rightarrow A_n$$

The verifier state transition corresponds to a set of concrete state transitions on the corresponding execution path:

$$S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_{n-1} \rightarrow S_n$$

Each concrete state S_i must be contained in the corresponding approximation A_i; otherwise, it is a verifier bug since attackers can manipulate such a concrete state, thereby breaking the verifier's conclusion. The key observation is that, to ensure the conclusion is correct, the verifier must over-approximate all possible concrete states at each program point. In other words, any concrete state not contained in the approximation can invalidate the conclusion. The property is fundamental and can be harnessed as an effective test oracle to validate the correctness of the verifier without requiring specifications. The ensuing challenge is how to determine whether or not concrete states are contained in the approximation.

State Embedding. This paper introduces state embedding, a novel and effective mechanism for validating the eBPF verifier. Our key insight is: one can effectively detect logic bugs by embedding a program with the aforementioned approximation-correctness checks that are expected to be validated by the verifier itself. State embedding contrasts pairs of programs, P and P'. Initially, a program P, accepted by the verifier, is executed to profile its concrete states. Next, P' is crafted by embedding sinks that contradict these observed states into P, challenging the verifier to validate it. A correct verifier should reject P' since a valid approximation must include the observed concrete states, thus accepting P' reveals a logic bug. More concretely, given an accepted program P and a profiled concrete state S. Corresponding to S is a variable A representing the verifier's approximation. P' is constructed by embedding the following program construct:

$$\textbf{if } S \in A \ \textbf{ then } \ \textit{verifier\_sink}()$$

The condition directs the verifier to check if the concrete state S is indeed contained within its approximation A, and *verifier_sink*() refers to any incorrect operation; encountering this during validation signals an error, indicating that S is correctly contained in A. One can easily realize the construct by utilizing the if-condition statement with equality comparison; we defer to Section 3.1 for concrete examples. Thus, P' can be applied to validate the verifier V: $\mathcal{P} \rightarrow \{safe, unsafe\}$, where marking P' as *unsafe* due to the triggered sink confirms the inclusion of S in A; conversely, deeming P' safe indicates a failure in capturing S within A, *i.e.,* a verifier's logic bug.

Realization. We realized state embedding as a practical tool, which we call SEV, and applied it in validating the eBPF verifier. First, for an eBPF program accepted by the verifier, SEV executes and profiles the program to gather its register states at each basic block. Second, to efficiently embed each state, we utilize the following optimization (which will be further elaborated in Section 3.1). At each basic block, a folding function is generated to fold the corresponding concrete register states into a global variable. A state-embedded program is synthesized by inserting the folding functions and embedding the concrete values of the global variables. Finally, the resulting program is used to validate the verifier; indeed, any failure to detect the sink indicates a logic bug in the verifier. Our evaluation results show that state embedding is highly effective. Within one month, we discovered 15 logic bugs exclusively in the verifier. This is a significant result considering that the verifier is primarily around 20,000 lines of code and, as aforementioned, has been partially verified. In addition, the verifier has gone through extensive security scrutiny and testing [7, 46]. Moreover, most of the bugs found are critical. For instance, two bugs are exploitable, where one allows users with `CAP_BPF` [41] to obtain root privilege and has existed for four years, and the other enables users with `CAP_PERFMON` [19] to obtain root privilege, both affecting kernel v5.10.33 and later.

State embedding significantly complements existing work. In comparison, our approach has several distinct advantages: (1) SEV treats the verifier as a grey-box rather than a black-box by inspecting the verifier states, which allows fine-grained detection of logic bugs; (2) by transforming and taking the state-embedded programs as input, the verifier automatically checks if the concrete states are contained, thus the approach requires little domain knowledge and is practical; (3) in general, one can easily embed rich concrete states, yet to detect all the sinks, the verifier must correctly collect approximations encompassing all the embedded states, thus being effective; and (4) state embedding can detect diverse logic bugs that lead to discrepancies between concrete states and approximations, *e.g.,* the bugs we found are located in various components including range analysis, stack access validation, *etc.* The key contributions of our work are:

- We propose state embedding, a novel and highly effective mechanism for detecting logic bugs in the eBPF verifier.
- We present SEV, a practical realization of state embedding, and apply it to stress test the eBPF verifier.
- We demonstrate state embedding's effectiveness by uncovering 15 previously unknown logic bugs in the eBPF verifier with 10 already fixed and many being critical.

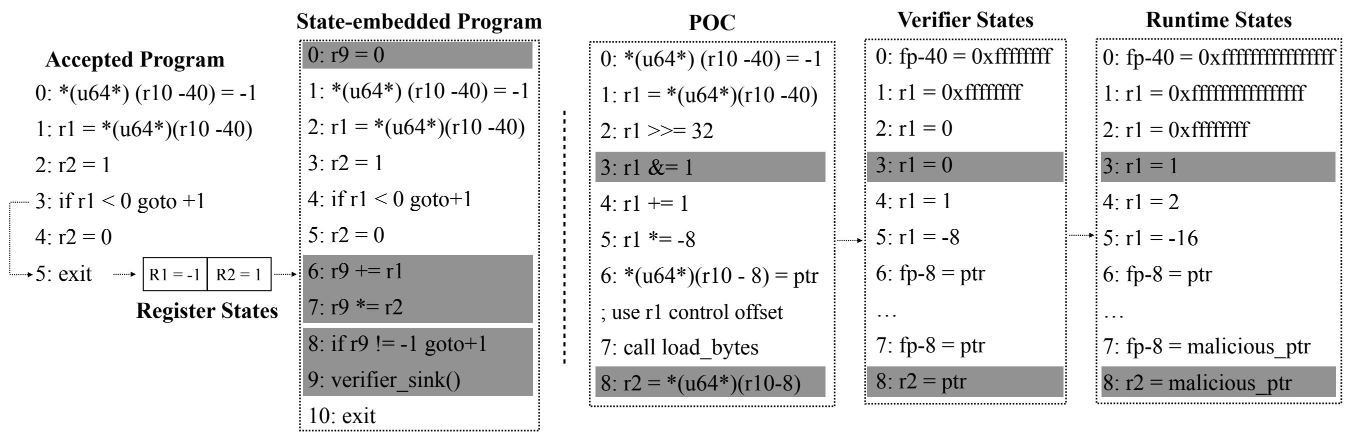

Figure 1: SEV performs state embedding by (1) profiling the concrete states of registers at each basic block, *e.g.,* R1 and R2; (2) folding R1 and R2 to R9 (#6 and #7); (3) embedding the concrete state of R9 (#8) and the verifier sink (#9). The instructions #8 and #9 implement the program construct and are the approximation-correctness check. The verifier interprets the if-condition #8 by validating if the runtime value -1 is within the approximation of R9, *i.e.,* determining if R9 could be -1, in which case the sink would be reported; otherwise, the verifier jumps from #8 to #10 and skips the sink. During validating the state-embedded program, the verifier skips the sink due to the logic bug. The root cause is sign information loss at #1. The POC program manipulates the state of R1 (#2 and #3), making the verifier believe R1 equals zero, whereas at runtime, it equals one. Consequently, the program overwrites the valid pointer stored on the stack with a malicious pointer (#6 and #7), thereby achieving arbitrary access.

2 Background and Illustrative Example

2.1 eBPF

eBPF is a register-based virtual machine that enables user space to extend the kernel dynamically. The user space first writes programs consisting of a sequence of eBPF instructions and loads the program with the `bpf()` system call. Programs operate on 11 registers (R0 to R10) and a fixed-size stack with four major types of instructions, namely load, store, arithmetic, and branch. An example program is presented in Figure 1. eBPF is adopted across different privilege levels, from unprivileged users [13] to those with certain capabilities [19, 41], and to fully privileged users. For instance, the `CAP_BPF` capability allows using eBPF with minimal privilege, widely applied in container scenarios. Consequently, the extensions are untrusted and a verifier is employed to validate their safety.

In a nutshell, the verifier traverses each execution path, interpreting every instruction in its abstract domains. Programs exhibiting any form of invalid behaviors, such as infinite loops and out-of-bounds access, are rejected. To strive for both soundness and precision, the verifier employs sophisticated algorithms to track program states. For instance, it gathers pointer types, register liveness, scalar ranges, *etc.* Scalar ranges are tracked using five abstract domains: four interval domains for different signs and bit-sizes, and the tristate number (`tnum`) domain [35] to model bit-wise operations. Scalar ranges are derived by combining information across these domains. The verifier models pointers based on region types and offsets, categorizing the former into more than twenty types, while tracking the latter using a variable. Furthermore, the verifier undergoes continuous updates by maintainers with new features and algorithms. The above features make the verifier the most complex component within the eBPF subsystem.

2.2 Illustrative Example

The key idea of our approach is to embed concrete states in programs such that when taking the state-embedded programs as input, we leverage the verifier to check whether the concrete states are contained in the approximation, thereby detecting logic bugs. In this section, we use an example to showcase how state embedding enables the detection of a subtle logic bug caused by a simple incorrect type cast. We also demonstrate the exploitation of the bug to highlight the significance of the verifier's correctness.

The Bug. The left segment of Figure 1 shows a valid eBPF program accepted by the verifier. Without effective test oracles, existing approaches would simply drop the case and proceed to the next iteration, since the program does not contain invalid behaviors. In comparison, SEV further utilizes state embedding to validate the verifier's approximation of the program, thereby uncovering this bug. Figure 2 shows the root cause and the patch proposed by us to fix the bug.

eBPF programs can spill registers or immediate values to the stack, and the verifier tracks the state of the stack accordingly. For the instruction `*(u64*)(r10-40)=-1` shown in the accepted program, the verifier marks the state of the accessed

```
diff --git a/kernel/bpf/verifier.c b/kernel/bpf/verifier.c
index 857d76694517..44af69ce1301 100644
--- a/kernel/bpf/verifier.c
+++ b/kernel/bpf/verifier.c
@@ -4674,7 +4674,7 @@ static int check_stack_write_fixed_off(...)
         insn->imm != 0 && env->bpf_capable) {
      struct bpf_reg_state fake_reg = {};

-     __mark_reg_known(&fake_reg, (u32)insn->imm);
+     __mark_reg_known(&fake_reg, insn->imm);
      fake_reg.type = SCALAR_VALUE;
      save_register_state(state, spi, &fake_reg, size);
   } else if (reg && is_spillable_regtype(reg->type)) {
```

Figure 2: A logic bug detected by SEV. During spilling immediate values on the stack, the verifier incorrectly casts the `i32` immediate value to `u32` type, thus losing sign information. We proposed this patch to drop the cast, since `__mark_reg_known()` accepts `u64` type, the compiler would correctly promote integer type and propagate sign information. The patch has been accepted in the upstream and back-ported to the stable kernels.

stack slot as a known scalar value. However, during this process, the verifier incorrectly casts the immediate value from `i32` to `u32` and then sets the state of the stack slot, which is `u64`, to the casted value, causing lost sign information and the state of the stack slot being updated to an incorrect value. As depicted in Figure 1, when storing `-1` to the stack, the verifier incorrectly marks the corresponding stack slot as a value whose higher 32 bits are all zero, *i.e.,* losing sign information. Subsequently, when loading the same stack slot back, the verifier state of the destination register does not match the original register, *i.e.,* the concrete value `-1` is not contained in the approximation. Figure 2 demonstrates the patch to fix the bug, which has been merged to the upstream and back-ported to the stable kernels, given its security impact.

In general, detecting logic bugs in the verifier poses significant challenges due to the following characteristics:

- **Hard to notice:** The abstract domains utilized by the verifier are complex and challenging to comprehend thoroughly, and the cyclomatic complexity of its tracking logic is high. To pinpoint logic bugs, one needs to precisely understand the verifier states and inspect them following the tracking logic. Conducting such a process is difficult, *e.g.,* the aforementioned incorrect type cast is likely to be overlooked.

- **Hard to detect:** Existing work treats the program under testing as a black-box, yet logic bugs of the verifier are likely to be silent errors. For instance, the behavior of the program demonstrated is correct, which simply accesses the stack within bounds and operates the registers, and in this sense, the verifier's conclusion seems justified. However, as shown by our approach, the verifier's approximation contains a subtle flaw, which is challenging for existing approaches to detect.

State Embedding. To enforce the safety of the program, the verifier must track the over-approximation of program states on each execution path, yet a concrete state not contained in the approximation can invalidate the conclusion. Based on the observation, our approach systematically transforms the program to embed concrete states within certain program constructs and utilizes the verifier to validate whether the aforementioned property holds during the checking process.

Step 1: The first step of state embedding is to profile concrete states, as illustrated in Figure 1. For a program accepted by the verifier, we execute and profile concrete states at each basic block of the execution path. Since the if-condition at #3 holds at runtime, the program jumps to #5, where we collect the concrete states of registers. The concrete states of R1 and R2 must be contained in the verifier's approximation.

Step 2: The second step performs state embedding. In practice, the collected state information is rich, and directly embedding each state makes the verifier fork and explore paths multiple times, potentially leading to redundant checks. We adopt folding to efficiently conduct state embedding. First, we initialize R9 (#0), a reserved register that holds the folded concrete state. Then, we generate a folding function at each basic block, which consists of arithmetic instructions that fold the collected states into the single register R9. The folding function in Figure 1 has two instructions #6 and #7, which fold R1 and R2 into R9. The value of R9 is calculated during the generation by evaluating those arithmetic instructions with the concrete states, which, in the example, equals `-1`. Finally, the instruction #8 and #9 implement the program construct, which embeds the folded concrete state with the if-condition instruction that compares R9 with its value `-1` and the verifier sink. The condition makes the verifier compare the folded state with its approximation of R9. The sink would be detected if the verifier deems `-1` is within the approximation.

Step 3: Subsequently, we can take the state-embedded program as input to the verifier to detect the logic bug. During the checking process, the verifier initially collects the program states from #0 to #4, where it incorrectly tracks the approximation of R1 (#1 and #2). Such an issue would be captured as the concrete state of R1 is folded into R9. Since the verifier determines `r1<0` not hold, it proceeds from #4 to #5. At #6 and #7, the verifier folds R1 and R2 into R9, where it erroneously concludes that the only concrete value within the approximation of R9 is zero. Therefore, the folded concrete state `-1` would not be contained in the approximation of R9, causing the verifier to skip the sink at #9. Consequently, we have found the logic bug in the verifier.

Our approach embeds concrete states within certain program constructs and utilizes the verifier to validate the approximations. State embedding provides two distinct advantages:

- **Fine-grained:** State embedding views the verifier as a grey-box rather than a black-box since it performs fine-grained validation on the verifier states. For the aforementioned example, which is overlooked by the existing approaches and even by the experienced maintainers, our approach further validates the approximations with the profiled concrete states.

- **Practical:** Our approach validates if the aforementioned property holds by embedding concrete states and utilizing the verifier to compare the states against the approximations. Therefore, state embedding requires little domain knowledge and is more practical.

The Exploit. The right segment of Figure 1 depicts a proof-of-concept (POC) that exploits the bug to obtain root privilege. In essence, the POC uses the bug to manipulate the verifier's knowledge about the program. Since the verifier incorrectly believes that the higher 32 bits of R1 are zero, which are in fact all one at runtime, the POC first constructs an evil register by right-shift and logic AND operations (#2 and #3). These operations make the verifier conclude R1 equals zero, while at runtime it equals one. Then, the POC stores a valid pointer on the stack and invokes a helper function that allows eBPF programs to load user space data to their stack. By using the evil register as the length parameter, the POC makes the verifier think that only 8 bytes are stored on the stack, while in fact it stores 16 bytes, thus overwriting the pointer with the user-controlled pointer. Finally, the POC achieves arbitrary access with the malicious pointer, while the verifier erroneously believes the program is operating the original valid pointer.

Since eBPF programs are executed in kernel space, the POC can perform various malicious operations, *e.g.,* overwriting credentials of the current task struct for privilege escalation. Our full POC enables users with `CAP_BPF` to achieve root access, and kernels v5.10.33 and later are affected. After we submitted the patches that fix the bug, we also received positive feedback from the maintainers of the eBPF subsystem:

> *"...I owe you a big thanks as well since this helps with our internal process. So thank you in advance!"*

It is important to highlight that the aforementioned exploit accomplishes all the malicious operations through the subtle bug, a simple incorrect type cast in one line of the verifier's code. In addition to the presented bug, we also found another exploitable bug allowing users with `CAP_PERFMON` to obtain root privilege, arising from incorrect tracking of memory accesses with variable offsets. The capability mechanism in Linux grants users the minimum privileges for specialized tasks, yet these bugs undermine this rule, posing significant security concerns. Notably, in the POC programs for both bugs, the number of instructions used to manipulate the verifier's knowledge is less than ten, further demonstrating the severity of the bugs found. This illustrates that our approach can detect critical logic bugs by leveraging the verifier to perform fine-grained validations on its approximations.

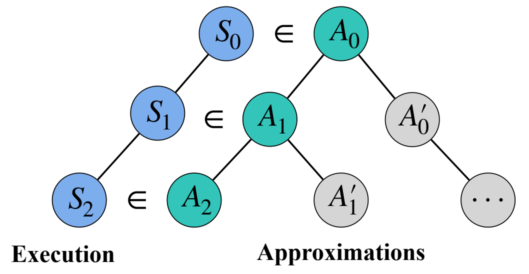

Figure 3: The verifier tracks program states on each possible path as shown in the right part, and the runtime execution corresponds to one of the paths. Each concrete state S_i must be contained in the approximation A_i; otherwise, operations on the non-contained states could be unsafe, *i.e.,* a logic bug.

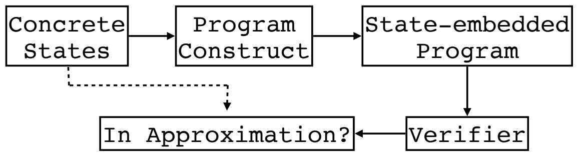

Figure 4: Our approach validates whether concrete states are contained in the approximation for logic bug detection by (1) embedding the concrete states in the program with the construct; (2) taking the state-embedded program as input to the verifier; and (3) leveraging the verifier to validate the states against the approximation.

3 State Embedding and SEV

In this section, we introduce state embedding, a folded variant, and describe our implementation of SEV.

3.1 State Embedding

State embedding is based on one fundamental observation: the concrete states must be contained in the corresponding approximations of the verifier, as shown in Figure 3. The goal of state embedding is to validate if this property holds during validation. As depicted in Figure 4, to achieve this, the approach executes and profiles an accepted program P, lacking inputs and external interactions, to collect concrete states, which remain consistent across multiple executions. Since those states are profiled from a real execution, they establish the ground truth that the corresponding approximations must properly contain them. Next, P is transformed to P' by embedding sinks that contradict those observed states. When taking P' as input, the verifier, following the same path as the real execution, compares the embedded concrete states against the approximations, thereby automatically validating its own correctness.

At the conceptual level, given a program P, a concrete state S, and the variable A holding the state (the approximation from the verifier's perspective), a state-embedded program P'

is synthesized by embedding the following program construct at the corresponding program location:

$$\textbf{if } S \in A \textbf{ then } \textit{verifier\_sink}()$$

The $\in$ in the program construct is an abstraction of the operators in the program, for which the verifier interprets to check whether the concrete state S is contained in the approximation A or not. The $\textit{verifier\_sink}()$ represents the operations, where the verifier reports errors. By embedding the program construct and taking P' as input, the verifier interprets the construct to check if the embedded state S is within A properly. The verifier sink in the construct is an indicator for the containment of the concrete state S.

Proposition 3.1 *State embedding does not introduce any invalid operations except for the sink.*

The aforementioned program construct does not introduce incorrect operations to the original accepted program except for the verifier sink. When interpreting the embedded construct, the verifier state A_i would be split into two states A_j and A_k, where the former follows the branch-taking path of the if-condition thus detecting the sink, and the latter continues the original execution path. Since the original program is considered safe under the approximation of A_i and $A_i = A_j \cup A_k$, *i.e.,* A_i is a superset of both A_j and A_k, the program is also considered safe with A_k. Therefore, the sink is the only expected error for a state-embedded program.

Corollary 3.1 *Failure to detect the sink indicates a logic bug in the verifier.*

More concretely, for a verifier V: $\mathcal{P} \rightarrow \{safe, unsafe\}$, since the original program P is considered safe by the verifier, $V(P') = unsafe$ with the sink being reported implies that the verifier correctly deems S is within the approximation A; on the contrary, $V(P') = safe$, *i.e.,* failure to detect the sink, indicates the concrete state S being missed from the approximation A, *i.e.,* a verifier logic bug.

Here we demonstrate the semantics of each part in the program construct with an example. In general, the verifier interprets concrete operators in the program in its abstract domain. Many concrete operators are available to implement the abstraction $\in$. For instance, as shown in Listing 1, one can utilize the if-statement with an == comparison to realize the construct; other comparisons, such as greater-equal (>=) or less-equal (<=), are also feasible, as per the implementation. The instruction writing to the read-only register R10 is an instance of the verifier sink in eBPF programs, and assertion failure can be used in other scenarios. After embedding, the verifier interprets the if-statement by validating if S is within the approximation of A to determine if the branch should be taken, in which case the sink would be reported.

```
...original program
The verifier checks if S is within A, when
interpreting the following if-statement.
if (S == A)
    verifier_sink();
...
```

Listing 1: Example of the program construct.

Folded Variant. A straightforward realization of state embedding is by profiling the program to collect concrete states and subsequently embedding the states with the corresponding program construct multiple times. However, directly embedding each state encounters two challenges: (1) the verifier may halt early at the first detected sink, leaving other sinks unchecked; and (2) the inserted sinks make the verifier fork exploring paths, introducing redundant checks. We propose a folded variant to embed the states in conjunction with one sink efficiently. For each concrete state in an execution:

$$S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_{n-1} \rightarrow S_n$$

we do not directly embed each S_i into the program. Instead, we generate a folding function f_i for each of them, which consists of simple computation operations, *e.g.,* ALU operations in eBPF programs, and hold the folded concrete state $\widehat{S}$ in a global variable $\widehat{A}$:

$$\widehat{S_1} = f_1(\widehat{S_0},\ S_1) \rightarrow \ldots \rightarrow\ \widehat{S_n} = f_n(\widehat{S_{n-1}},\ S_n)$$

Finally, we embed the folded state $\widehat{S_n}$ once with the following program construct:

$$\textbf{if } \widehat{S_n} \in \widehat{A} \textbf{ then } \textit{verifier\_sink}()$$

Since only the folded state is embedded, the verifier forks once. The bug-detecting capability of this variant is equivalent to the original form since incorrect approximations are likely to be propagated to the approximation of the global variable during the continuous testing campaign. The variant provides further benefits, *e.g.,* to detect the sink, the verifier needs to not only correctly track states for the original program, but also properly simulate the folding functions.

3.2 The SEV Implementation

We applied state embedding to detect logic bugs in the eBPF verifier. Algorithm 1 illustrates the major workflow. In each testing iteration, SEV first utilizes the program generator that we developed to obtain eBPF programs accepted by the verifier (lines 4-6), and then executes and profiles the program to collect the register states at each basic block (line 7). We follow established compiler testing methodologies [31] by utilizing deterministic, closed eBPF programs that require only a single round of profiling and the profiled states are consistent across executions. Based on the concrete states,

Algorithm 1: Workflow of SEV

```
1  Procedure Validate():
2      $LogicBugs \leftarrow \emptyset$
3      while not terminate do
4          $P \leftarrow$ NextProg()
5          if Verify($P$) = reject then
6              continue
           // Profile states at basic block
7          $S \leftarrow$ Profile($P$)
           // Embed the concrete states
8          $P' \leftarrow$ StateEmbedding($P$, $S$)
           // Validate the eBPF verifier
9          if Verify($P'$) = accept then
10             $LogicBugs \leftarrow LogicBugs \cup P'$
```

Algorithm 2: State Embedding

```
1  Function StateEmbedding(Program P, States S):
2      $FoldedState \leftarrow$ Initialize()
       // Basic block to folding function map
3      $FoldingFns \leftarrow \emptyset$
4      foreach $BB$, $Regs \in S$ do
5          $F \leftarrow FoldingFns[BB]$
6          if $F$ not exists then
               // Generate folding function
7              foreach $Reg \in Regs$ do
8                  $F \leftarrow$ GenALU($F$, $Reg$)
           // Update Folded State
9          $FoldedState \leftarrow F(Regs, FoldedState)$
       // Insert the folding functions
10     $P' \leftarrow$ InsertFoldingFns($P$, $FoldingFns$)
       // Embed the folded state
11     $P' \leftarrow$ EmbedFoldedState($P'$, $FoldedState$)
12     return $P'$
```

SEV transforms the program to inject folding functions at each basic block and embed the folded concrete state (line 8). Finally, the state-embedded program is used as input to the verifier (line 9), and a logic bug is detected if the program is accepted (lines 10-11), *i.e.,* the verifier is incapable of detecting the sink. The aforementioned loop continuously tests the verifier with state-embedded programs.

SEV first needs to obtain eBPF programs that can pass the verifier. We devise a program generator to facilitate state embedding, adhering to the established program generation approaches [50]. First, we ensure the instruction encoding and the control flow of a generated program are valid by following the instruction specification [6] and representing a program as a structured graph, thereby avoiding being rejected early. Next, we synthesize the program by combining several basic structures, *e.g.,* if-else block and back-edge, and leveraging lightweight global state information, such as register and stack slot states, to generate instructions reflecting realistic usage patterns. We categorize a register state into several types, including uninitialized, scalar value, and pointer, and synthesize operations accordingly, *e.g.,* generating pointer accesses or offset operations if a register stores a pointer. The generator continuously provides programs for the testing campaign.

We implemented a tracer based on the existing kernel infrastructures to profile programs. The tracer intercepts the execution of programs at each basic block and captures the instruction index and the register states, which are appended into an internal state buffer. The tracer interface is exposed via a virtual device so that the user space can utilize the functionality flexibly and access the buffer via `mmap()` for shared memory. For each accepted program, SEV executes it with the tracer enabled and decodes the buffer in user space.

Algorithm 2 presents SEV's implementation of state embedding. The inputs are the accepted program and the concrete states at each basic block collected with the tracer. We first initialize the folded state (line 2), and R9 is reserved to ensure its availability for storing this value. The algorithm maintains a map associating each basic block with its folding function (line 3). The collected states are essentially a basic block trace in conjunction with the states, where the basic block could appear multiple times due to loops. By using the map, the algorithm ensures the folding function is generated once for each basic block. The folding function is generated by synthesizing various eBPF arithmetic instructions for each collected register (lines 6-8). We consider all the ALU instructions the verifier can accurately track, excluding the division and all unary operations. Folding functions are generated by randomly selecting those operations applied to non-zero registers, preventing the state from being easily reduced to zero. The folded state is updated by calculating the folding function with the concrete states (line 9). Finally, the state-embedded program is generated by inserting each folding function in the corresponding basic block and embedding the folded state with the instructions (lines 10-11), as shown in Listing 2.

```
if r9 != FoldedState goto+1
r10 = 0
```

Listing 2: The instructions used for embedding the folded state and the verifier sink in eBPF programs.

The first instruction in Listing 2 makes the verifier validate whether the folded state is contained in the approximation of R9. The second instruction is the verifier sink, an illegal operation where the verifier reports a "writing to the read-only register R10" error. The sink is skipped if the folded state is not contained, indicating a logic bug. The state-embedded program can thus be used for validating the verifier.

4 Evaluation

In this section, we evaluate the effectiveness of state embedding by applying SEV to detect logic bugs in the eBPF verifier. Highlights of our results are as follows:

- **Considerable bugs:** We have found 15 previously unknown logic bugs solely in the eBPF verifier.
- **Diverse bug types:** The root causes of the bugs are various and located in different components of the verifier.
- **Critical severity:** Most of the bugs found by SEV are critical, posing various security implications.

We believe that the quantity and quality of the bugs found by our prototype SEV have demonstrated the effectiveness of state embedding in uncovering logic bugs in the verifier.

4.1 Evaluation Setup

Environment. All the bug-finding experiments were conducted on a Linux server with a 64-core AMD Ryzen Threadripper 3990X Processor, where each core has two threads, and the memory size of the server is 256 GiB. The version of the host Linux kernel is v5.15. To ensure that these experiments did not affect the host system, we conducted our testing within multiple virtual machine instances. These instances were created using QEMU version 6.2.0, with KVM employed to provide acceleration. The guest environment in each instance consisted of a minimal Debian distribution disk image, and the system was booted using the compiled kernels. In addition to incorporating common kernel configurations, we also enabled the eBPF subsystem-related options [5].

Kernel Version. We chose the eBPF upstream repository for testing, and the reasons are as follows: (1) the uncovered logic bugs in the upstream are likely to be previously known, and thus should be fixed immediately; (2) testing upstream enables the detection of bugs that may impact various past stable versions; and (3) testing the upstream kernel prevents newly introduced bugs being merged into subsequent releases. In addition to the built configuration previously mentioned, we also patched the kernel and enabled related options since we modified the eBPF interpreter `bpf/core.c` to intercept the execution of eBPF programs for state tracing.

Testing Process. SEV is designed to automatically execute the entire testing process. Initially, after configuring the disk image and the kernel for testing, SEV determines the appropriate QEMU command line and subsequently initiates the virtual machine using the specified kernel. Upon successful booting of the virtual machine, SEV proceeds to the testing phase. This involves the generation of eBPF programs and the validation of the verifier using state-embedded programs. A shared directory is established to transfer the testing results between the host and the guest. Subsequently, SEV checks the liveness of the virtual machine and restarts the whole campaign if it detects the system is not alive. We utilized SEV to test the eBPF upstream with the aforementioned testing process for one month.

Bug Triage. We triage and deduplicate all the bugs found based on their root causes. In principle, any failure to detect the sink in the state-embedded programs indicates a logic bug in the verifier. When SEV reports such a case, we further inspect it to locate the root cause. During the testing campaign, SEV retains the original program, the captured runtime states, and the corresponding state-embedded program when a logic bug is identified. We inspect the programs to pinpoint the instruction where the approximation of the verifier mismatches the runtime states, and then analyze the preceding instructions to collect related operations that produce the operands for the culprit instruction. The culprit instruction in conjunction with related operations enables us to locate the incorrect verifier logic. Finally, one can look into those parts of the verifier and analyze the root cause.

4.2 Quantitative Results

Bug Number. We applied SEV to test the eBPF verifier for one month and triaged the discovered bugs based on their root causes as mentioned in Section 4.1. Note that we only reported unique bugs to the eBPF mailing list, and only bugs with different root causes are counted in our evaluation. As a result, we have found 15 unique logic bugs in the eBPF verifier within one month, of which 10 have been fixed at the time of paper submission. The number of found bugs is significant considering: (1) the codebase of the eBPF verifier is relatively small compared to other subsystems, *e.g.,* it mainly contains 20,000 lines of code; (2) the verifier has undergone thorough security scrutiny by the community and is one of the most extensively tested components in the kernel, *e.g.,* eBPF self-tests [7] contain a large set of programs covering different corner cases to test the verifier; and (3) previous research efforts have applied verification on parts of the eBPF verifier [44, 45, 52]. The aforementioned results demonstrate that our approach is highly effective in uncovering logic bugs.

Furthermore, compared to manual testing, which requires substantial effort to keep up with the development of eBPF, our approach can continuously test the eBPF verifier along with its frequent updates and modifications. Existing testing tools like Syzkaller can uncover memory issues in the eBPF subsystem, yet they encounter difficulties in logic bug detection due to the lack of effective test oracles. While formal verification provides a strong guarantee, synthesizing specifications for the verifier is inherently complex, whereas SEV utilizes state embedding to automatically uncover logic bugs in the verifier without requiring specifications. Therefore, state embedding uniquely complements existing approaches.

Table 1: Number of bugs found by SEV in different locations.

Bug Location	Range Analysis	Memory Access	State Prune	CFG Check	Other
#	6	3	2	2	2

Bug Types. In general, logic bugs in the verifier can be classified into two categories: (1) incorrectly accepting unsafe programs, *i.e.,* soundness bugs, and (2) incorrectly rejecting safe, correct programs, *i.e.,* completeness bugs. The former introduces potential security issues, *e.g.,* allowing malicious programs to be loaded, while the latter forces developers to heavily refactor their programs to mitigate the verifier's imprecision. Overall, we have found 12 soundness bugs and three completeness bugs. Table 1 shows the bug details.

Our approach can detect various soundness bugs in the verifier for the following reasons. eBPF supports four major types of instructions, including load, store, ALU, and jump operations. For ALU and jump, the eBPF verifier mainly performs range analysis and pointer arithmetic checks. Since these operations are conducted on the registers, logic bugs in those components mainly result in incorrect approximation of register states. SEV can directly detect those logic bugs because our approach validates if the concrete states are contained in the approximation. For example, six of the bugs found are related to range analysis. For logic bugs in load and store checks, although we do not profile all the memory accessible to eBPF programs, SEV can still detect bugs in these areas because incorrect tracking on those states can be propagated to the approximation of registers. For instance, three of the bugs found are related to memory access validation, including the incorrect stack spill checking illustrated in Figure 1. Furthermore, we also uncovered logic bugs in other components, *e.g.,* the state pruning procedure.

SEV can also uncover completeness bugs as an additional design benefit. As detailed in Section 3, state embedding does not introduce any incorrect operations except for the sink. If the verifier rejects state-embedded programs for reasons other than the sink, it indicates the presence of a completeness bug. More broadly, the idea of state embedding can be specifically tailored for completeness bug detection. For example, replacing the sink with valid operations in the transformed programs would ensure their correctness, thereby highlighting any inaccuracies in the verifier's rejection of these programs. Completeness bugs not only interfere with development but also reflect potential implementation issues in the verifier. SEV uncovered three such bugs, where two are related to control flow graph checking, and the other is inconsistent stack access validation. While these bugs may not directly pose security issues, they significantly impact the usability of eBPF. For instance, one bug we discovered causes the verifier to erroneously reject a set of programs due to a specific control flow pattern, despite these programs being correct.

These results underscore the effectiveness of state embedding in identifying a diverse array of logic bugs, and generally, state embedding can detect logic bugs that result in a divergence between the approximations and the concrete states.

Bug Impact. Logic bugs within the eBPF verifier hold critical implications. All of the 15 bugs found by SEV are located within the verifier, specifically in `verifier.c`, and most of them are critical. We have demonstrated that two of the found bugs are exploitable, and each can be exploited to achieve local privilege escalation. Beyond these, other uncovered bugs have diverse implications. For example, some bugs in the range analysis can circumvent the verifier's enforcement that the return values of certain programs must be within specified ranges, thereby potentially affecting the caller. The bugs in the state pruning procedure can be used to load programs containing infinite loops, leading to system hangs.

4.3 Assorted Bug Samples Found by SEV

To further demonstrate the characteristics of the uncovered bugs, we highlight several examples in this section.

Figure 5a: The range analysis is an important component of the verifier since it is the foundation for various safety checks, *e.g.,* memory access check and control flow validation. Figure 5a shows a logic bug found by SEV in the range analysis, where the verifier incorrectly tracks registers' states after simulating the fall-through path of the branch condition. The root cause is the verifier's inability to correctly handle the `JSLE` instruction when comparing a range with a non-overlapping constant. More concretely, R9 is initialized at first and updated subsequently, and the range of R8 and the value of R4 are non-overlap at #6, after which the verifier incorrectly marks R8 and R9 as constant values. After the arithmetic operations (#7 and #8), the runtime value of R9 is one at #9, which differs from the verifier's approximation, a constant zero. The bug can break the verifier's restriction. For instance, the verifier enforces that the return value of certain program types can only be zero to not modify the caller states, yet the bug leads to programs with arbitrary return values being loaded. After we reported the bug, the maintainers also enhanced the return value-checking logic in the verifier.

Figure 5b: The verifier allows privileged users to load programs with back-edges and detects and rejects infinite loops during checking. SEV uncovered a logic bug in the verifier that incorrectly rejects programs with well-defined structures. For the program shown in Figure 5b, the verifier rejects it, reporting an incorrect back-edge from #3 to #4. However, such an error is not accurate, since #3 to #4 is not a back-edge. Furthermore, the behavior of the program is correct as the loop in the program is bounded. The root cause is that the control flow-checking procedure of the verifier is incapable of handling the structure pattern of the program. The bug

```
0: (b7) r9 = -2                    ; R9=-2
1: (37) r9 /= 1                    ; R9=scalar()
2: (bf) r8 = r9                    ; R9=scalar(id=1) R8_w=scalar(id=1)
3: (56) if w8 != 0xfffffffe goto pc+4 ; R8=scalar(var_off=(0xfffffffe;
                                        0xffffffff00000000))
4: (65) if r8 s> 0xd goto pc+3     ; R8=scalar(smax=13)
5: (b7) r4 = 2                     ; R4=2
6: (dd) if r8 s<= r4 goto pc+1     ; R4=2 R8_w=0xfffffffe
7: (cc) w8 s>>= w9                 ; R9=0xfffffffe R8=scalar()
8: (77) r9 >>= 32                  ; R9=0
9: (57) r9 &= 1                    ; R9=0
10: (95) exit
```

(a) A bug in the verifier's range analysis.

```
0: (b7) r4 = 0x35
1: (b7) r8 = r4
2: (05) goto+2
3: (1f) r9 -= r4
4: (1f) r9 -= r8
5: (0f) r8 += r4
6: (a6) if r8 < 0x64 goto-4
7: (bf) r0 = r9
8: (95) exit
```

(b) A bug in the CFG checking.

```
0: (bf) r0 = r10                ; R0=fp0
3: (18) r5 = 0x1d00000025       ; R5= 0x1d00000025
2: (bc) w9 = w0                 ; R9=scalar(var_off=(0x0; 0xffffffff)
3: (47) r9 |= -12               ; R9=scalar(var_off=(…; 0xb))
4: (0f) r9 += r0                ; R9=fp(off=0, u32_min=-12)
5: (76) if w5 s>= 0xfffffff6 goto pc+16  ; R5=0x1d00000025
6: (72) *(u8 *)(r9 -221) = -19    ; stack_depth=221
7: (95) exit
```

(c) A bug in the stack depth tacking.

```
0: (18) r4 = map_ptr
1: (18) r1 = 0x1d
2: (55) if r4 != 0x0 goto pc+4
3: (1c) w1 -= w1
4: (18) r9 = 0x32
   (00) reserved_code
5: (56) if w9 != 0xfffffff4 goto pc-2
6: (95) exit
```

(d) A bug in the jump target checking.

Figure 5: Assorted eBPF programs that trigger logic bugs. The left part of each sub-figure shows the eBPF instructions and the content after each semicolon presents the verifier's approximations, illustrating the bug cause. `scalar()` and `fp()` show that the tracked value is a scalar and stack pointer, `var_off()` is the `tnum` domain, and `stack_depth` is the tracked depth of used stack.

causes all the correct programs with this structure pattern to be rejected, thus requiring heavy code refactoring to mitigate the verifier's bug. The inserted folding functions triggered the bug, demonstrating the additional advantages of the folded variant of state embedding. The bug has been fixed and the patch was back-ported to the stable kernels.

Figure 5c: The verifier tracks the stack depth of the program and uses this information to subsequently allocate the stack area before execution, thus the correctness of the calculated stack depth is important. However, SEV uncovered that the verifier incorrectly overlooks the variable offset in stack accesses, leading to the collected stack depth being smaller than the size that the program may access at runtime. As illustrated in Figure 5c, at first, R9 is a scalar with a minimum value of -12, and the tracked range information is correct. The instruction #4 adds the stack pointer to R9, thereby making R9 a stack pointer with a variable offset. The verifier incorrectly marks the stack depth of the program as 221 at the stack writing instruction #6 without considering the variable offset of R9, *i.e.,* its possible minimum value. The bug has existed for four years and is exploitable for privilege escalation.

Figure 5d: Most instructions of eBPF adopt basic encoding with an eight-byte length, while a special kind of load instruction uses the wide instruction encoding and appends a second eight-byte immediate, the code of the second part is the reserved code. The target of jump instructions in eBPF programs must be within bounds and be a valid instruction. However, SEV detected a logic bug in the verifier that reports programs containing invalid jump targets with an incorrect reason. As depicted in the figure, the instruction #4 is a special load that uses the wide instruction encoding and its second part contains the reserved code. The program incorrectly jumps from #5 to the second part of #4, *i.e.,* jumping into the reserved code, yet the verifier incorrectly reports that the program contains an invalid load instruction. The root cause of the bug is that the verifier overlooks the special case while checking the jump target.

```
0: (18) r9 = 0x00000018         ; R9= 0x18
1: (85) call get_cgroup_id#123 ; R0=scalar()
2: (5c) w9 &= w0                ; R9=var_off(…;0x18)
3: (b5) if r0 <= 0xfffffffb goto pc+3
                                ; R0=var_off(…;0x3)
4: (5d) if r9 != r0 goto pc+2   ; R0=-4 R9=-4
5: (c7) r9 s>>= 23              ; R9=-1
6: (95) exit
```

Figure 6: A logic bug in the verifier.

Figure 6: SEV also uncovered that the range analysis of the verifier is incapable of correctly handling equality comparison when the ranges of two operands do not overlap. This issue is exemplified in Figure 6. At #2 the mask of R9 is 0x18, and the verifier determines all the bits of R9 are known to be zero except for the fourth, and the fifth bit (unknown). The mask of R3 is 0x3, where only the lower two bits are unknown and all the other bits are one, and thus the range of R9 and R3 are non-overlap. However, the verifier erroneously assigns both R0 and R9 as -4 after the if-condition with equality comparison at #4, where the runtime value of R9 consistently remains 0x18, *i.e.,* a logic bug in the verifier.

4.4 Throughput Impact

SEV synthesizes state-embedded programs by profiling runtime states using the tracer mentioned in Section 3 and subsequently integrating the generated folding functions and embedding the folded concrete state. In this section, we evaluate the performance impact of SEV on three key aspects: the influence of profiling on program execution speed, the effect of state embedding on the verification time, and the overall impact on testing throughput.

The evaluation procedure is as follows. First, SEV is utilized to continuously generate programs. For accepted programs, we execute them both with and without the tracer and compare the execution times in both scenarios to determine the impact of the profiling. Subsequently, we embed the concrete states and measure the difference in verification time between the state-embedded program and the original program. To account for any nondeterministic factors, the above process is repeated multiple times for each program. Finally, to ascertain the impact on testing throughput, we run SEV with and without the profiling and state embedding. All the comparisons are carried out over a 24-hour period, and the average results are compiled and reported.

The evaluation results show that the average impact of profiling on program execution is 5.3%. This relatively minor impact is primarily due to the profiling being performed at the basic block level, where the tracer efficiently appends the states to a shared buffer. In terms of the verification time, the impact of state embedding results in an average increase of 17.2%. The minimal complexity added to the original program by state embedding, which involves arithmetic operations and a single embedding of the sink, contributes to this increase. Notably, the overall impact on testing throughput is only 1.6%. This lower impact, compared to program execution and verification time, is because state embedding is conducted only after programs are accepted, an infrequent event. In addition, the embedding constitutes a small part of the entire testing campaign, which also includes the generation, test case persistence, *etc.* In summary, we conclude that state embedding imposes a reasonable overhead and the impact is well within expectations, given its ability to uncover logic bugs.

4.5 Discussion

Coverage Impact. Coverage in our context involves two aspects: (1) in the generated programs (raw coverage), and (2) in the verifier (induced coverage), where the former may affect the profiling stage and the latter is related to the testing sufficiency. To ensure a stable raw coverage, we adopt closed eBPF programs as mentioned in Section 3.2 that require only a single round of profiling. For the induced coverage, we cover the verifier's functionality with the program generator, and optimizing the induced coverage is orthogonal to this work.

False Positives/False Negatives. As illustrated in Section 3, in principle, state embedding does not introduce any invalid operations to the original program except for the embedded sink. In practice, our approach has not resulted in any false positives, *e.g.,* the found bugs pose certain security implications. Being a testing technique, our approach can suffer from false negatives. The major reason is that state embedding requires programs accepted by the verifier, yet the generator may not be able to explore a diverse, thorough search space. Our main goal in this work is to detect a wide range of logic bugs with the principled idea, *i.e.,* concrete states being contained in the approximation. In addition, the inserted arithmetic instructions may hinder some non-contained states, albeit with a low likelihood. SEV operates within a continuous testing loop, which therefore inherently enhances the detection of such anomalies over time, even if a specific combination of state values momentarily evades detection.

Verifier Changes. The eBPF subsystem is undergoing continuous updates, incorporating new features as it develops. Despite these changes, state embedding remains widely applicable and is not limited to specific static checkers. Our implementation, importantly, does not depend on the internal workings of the verifier. This is because SEV mainly transforms the accepted programs to embed concrete states, after which the program is delivered to the verifier for validation. Changes within the verifier, such as new abstract domains, primarily affect how the state-embedded program is validated, while the transformation remains unaffected.

State Pruning. The state pruning procedure [12] in the verifier evaluates the current state against the known safe states to determine redundancy, thereby pruning explored paths. However, this technique cannot be applied to eliminate extra paths for direct embedding. When comparing states, the verifier performs a detailed analysis of registers marked as precise. The registers of inserted sinks are marked precise, yet their value ranges vary between the branch-taken and fall-through paths, making them unprunable. Folding addresses this by embedding the folded state once at the end of the program.

5 Related Work

eBPF Verification. Existing work [16] applies formal verification to several components of the eBPF verifier, significantly advancing the verifier's correctness. For instance, recent efforts [52] have proved the soundness of the implementation of the tristate number in the verifier, and we did not detect any bugs in the corresponding location, *i.e.,* `tnum.c` in the kernel. However, `tnum` is a small component of the verifier, which mainly contains 200 lines of code. Agni [45] is more ambitious, aiming to verify the range analysis of the verifier, which consists of 2,100 lines of code. The tool automatically converts the C source code to SMT formulas and utilizes SMT solvers to detect discrepancies between the specification and the implementation. The work concludes that the tool proved the soundness of range analysis in Linux v5.19, the latest version when the work was conducted. Beyond reasoning about the correctness of the verifier, Jitterbug [34] applies automated verification on the eBPF JIT compiler.

Nevertheless, devising specifications for the eBPF verifier automatically or manually is challenging and requires deep domain knowledge. Consequently, the synthesized specifications can be incomplete or inconsistent with the implementation. For example, while Agni aims to perform verification on the verifier's range analysis, the generated verification conditions are incomplete [4, 9, 10, 14]. Indeed, we identified logic bugs in this component within the same kernel version. In comparison, state embedding has the following unique advantages: (1) our approach does not require predefined specifications, but utilizes state embedding to perform fine-grained checks for logic bug detection; (2) state embedding is capable of testing various components not just the range analysis, *e.g.,* SEV also uncovered bugs in the memory access validation; and (3) since we leverage the verifier to check whether the concrete states are contained in the approximation, our approach is agnostic to the verifier's internal and can be applied along with its fast changes. Therefore, our approach significantly complements the existing work.

Static Analyzer Testing. Some work [27, 53] proposes to detect bugs in static analyzers by collecting the information of both analyzers and programs and directly comparing them. For example, Wu et al. [49] propose to profile pointer alias at runtime and compare the information directly with the knowledge from the alias analysis algorithm. Similarly, Buzzer [26] extracts the verifier log, conducts an offline comparison between the log information and the collected map value, and uncovered one logic bug. However, this direct comparison approach requires: (1) collecting and parsing both the runtime information and the analyzers' states, and (2) correctly conducting the comparison, which requires substantial domain knowledge to interpret the verifier states and is coupled with concrete implementations. In comparison, SEV only profiles program states at each basic block and utilizes state-embedded programs to enable the verifier to conduct the validation, thus being efficient and agnostic to verifiers. In addition, Bugariu et al. [17] propose to test abstract domains by interactively invoking operations and using the mathematical properties of the domains as test oracles. However, applying this approach in the eBPF verifier is challenging: (1) the verifier is integrated into the kernel and does not support interactively invoking its internal operations, and (2) the eBPF verifier does not provide a specification and mixes up the abstract operators in one domain with the refining operations [45], thus the mathematical properties summarized are not directly applicable in this context.

Differential Testing. α-Diff [29] conducts differential testing between several static analyzers to identify logic bugs in them. However, this approach hinges on the precondition that the precision of static checkers is comparably high and that reference implementations are well-established. Except for the verifier in Linux, Gershuni et al. [22] propose a potential reference verifier. Yet, it lacks support for some important features, *e.g.,* various modes of basic ALU instructions, and experiences precision issues [11]. Therefore, using the tool for differential testing would be ineffective in detecting logic bugs in the kernel verifier. Furthermore, our approach differs significantly from differential testing methods and is an instance of metamorphic testing. α-Diff relies on multiple checkers and does not provide ground truth for each variant it produces. In contrast, state-embedded programs contain the ground truth by construction, eliminating the need for other reference verifiers.

Assertion Generation. Existing work aims to automatically generate assert statements to detect logic bugs [48, 51], establishing input-output relationships using program analysis or deep learning. In contrast, our approach does not aim to assert input-output relations, which is nontrivial for the verifier's logic bug detection, but to validate a core property: the concrete states must align with the verifier's approximations. It embeds concrete states into the program, leveraging the verifier to conduct the containment checks. Therefore, state embedding fundamentally differs from assertion generation, in terms of both intent and methodology.

Kernel Fuzzing. Fuzz testing [21, 30, 38] is an effective approach and has been applied in kernel scenarios. For instance, Syzkaller [47] is capable of testing the eBPF subsystem by invoking the `bpf()` system call with random arguments. It has been integrated into upstream testing and has uncovered many memory errors [46]. Similarly, another work by iovisor [25] employs libfuzzer [36] to generate random byte sequences for testing the verifier. BVF [42] captures memory errors in eBPF programs with sanitation to indirectly detect correctness bugs. Nevertheless, existing fuzzers [33] highly rely on sanitizers to capture bugs [39, 40], and they experience difficulties in the verifier's logic bug detection. Therefore, our approach complements the existing fuzzing work and state embedding can be utilized for direct logic bug detection.

6 Conclusion

In this paper, we have introduced state embedding, a novel and effective technique for logic bug detection in the eBPF verifier. Our approach systematically transforms the program to embed concrete states. The state-embedded program can subsequently be used to test the verifier by validating whether or not concrete states are contained in the approximation of the verifier. By applying state embedding in testing the eBPF verifier, our prototype SEV has successfully uncovered 15 logic bugs—many are critical, and two are exploitable for local privilege escalation—demonstrating the effectiveness of our approach.

Acknowledgments

We thank the anonymous OSDI reviewers and our shepherd, Ken Birman, for their valuable feedback on the earlier versions of this paper. Furthermore, we are grateful for the comments and suggestions from our colleagues at ETH Zurich.

References

[1] CVE-2021-3490. https://nvd.nist.gov/vuln/detail/CVE-2021-3490.

[2] CVE-2021-4159. https://nvd.nist.gov/vuln/detail/CVE-2021-4159.

[3] CVE-2022-23222. https://nvd.nist.gov/vuln/detail/CVE-2022-23222.

[4] Divergence between Specification and Implementation in Agni. https://github.com/bpfverif/agni/issues/12. Accessed: Nov 20, 2023.

[5] eBPF Build Config. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/tools/testing/selftests/bpf/config?id=0dd3ee311255.

[6] eBPF Instruction Set. https://www.kernel.org/doc/html/next/bpf/instruction-set.html.

[7] eBPF Selftests. https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf-next.git/tree/tools/testing/selftests/bpf.

[8] eBPF Verifier. https://docs.kernel.org/bpf/verifier.html.

[9] False Negatives and Incomplete Specifications in Agni. https://github.com/bpfverif/agni/issues/15. Accessed: Nov 20, 2023.

[10] Incapable of Generating Formulas for Various Kernel Versions in Agni. https://github.com/bpfverif/agni/issues/10. Accessed: Nov 20, 2023.

[11] PREVAIL Issues. https://github.com/vbpf/ebpf-verifier/issues. Accessed: Nov 20, 2023.

[12] State pruning in the eBPF verifier. https://docs.kernel.org/bpf/verifier.html#pruning.

[13] Unprivileged bpf(). https://lwn.net/Articles/660331/.

[14] Verification Not Reach Completion after Prolonged Time in Agni. https://github.com/bpfverif/agni/issues/13. Accessed: Nov 20, 2023.

[15] Andrea Arcangeli. Seccomp BPF (SECure COMPuting with filters). https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html.

[16] Sanjit Bhat and Hovav Shacham. Formal Verification of the Linux Kernel eBPF Verifier Range Analysis. https://sanjit-bhat.github.io/assets/pdf/ebpf-verifier-range-analysis22.pdf, 2022.

[17] Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, and Peter Müller. Automatically Testing Implementations of Numerical Abstract Domains. In *Proceedings of ASE '18*, page 768–778, 2018.

[18] Paul Chaignon. Cyclomatic Complexity of the eBPF Verifier. https://pchaigno.github.io/ebpf/2019/07/02/bpf-verifier-complexity.html, 2023.

[19] Jonathan Corbet. CAP_PERFMON. https://lwn.net/Articles/812719, 2020.

[20] 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 POPL*, page 238–252, 1977.

[21] Google developers. OSS-fuzz: Continuous Fuzzing of Open Source Software. https://github.com/google/oss-fuzz, 2017. Accessed: Nov 20, 2023.

[22] 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 PLDI*, page 1069–1084, 2019.

[23] Tejun Heo. sched: Implement BPF Extensible Scheduler Class. https://lwn.net/Articles/916290/.

[24] William E. Howden. Theoretical and Empirical Studies of Program Testing. In *Proceedings of ICSE*, page 305–311, 1978.

[25] iovisor. bpf-fuzzer: Fuzzing Framework Based on libfuzzer and Clang Sanitizer. https://github.com/iovisor/bpf-fuzzer.

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

[27] Timotej Kapus and Cristian Cadar. Automatic Testing of Symbolic Execution Engines via Program Generation and Differential Testing. In *Proceedings of ASE*, page 590–600, 2017.

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

[29] Christian Klinger, Maria Christakis, and Valentin Wüstholz. Differentially Testing Soundness and Precision of Program Analyzers. In *Proceedings of ISSTA 2019*, page 239–250, 2019.

[30] lcamtuf. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/, 2013.

[31] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler Validation via Equivalence Modulo Inputs. In *Proceedings of PLDI*, page 216–226, New York, NY, USA, 2014. Association for Computing Machinery.

[32] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-Level Packet Capture. In *Proceedings of USENIX'93*, page 2, 1993.

[33] Mohamed Husain Noor Mohamed, Xiaoguang Wang, and Binoy Ravindran. Understanding the Security of Linux EBPF Subsystem. In *Proceedings of APSys*, page 87–92, 2023.

[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 *Proceedings of OSDI*, 2020.

[35] Jan Onderka and Stefan Ratschan. Fast Three-Valued Abstract Bit-Vector Arithmetic. page 242–262, 2022.

[36] LLVM Project. libFuzzer: a Library for Coverage-guided Fuzz Testing. https://llvm.org/docs/LibFuzzer.html.

[37] Jay Schulist, Daniel Borkmann, and Alexei Starovoitov. Linux eBPF. https://ebpf.io.

[38] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *USENIX Security*, pages 167–182, August 2017.

[39] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of USENIX ATC*, page 28, 2012.

[40] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, page 62–71, 2009.

[41] Alexei Starovoitov. CAP_BPF. https://lwn.net/Articles/820560, 2020.

[42] 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 EuroSys*, page 689–703, 2024.

[43] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications. *ACM Comput. Surv.*, 53(1), feb 2020.

[44] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. Sound, Precise, and Fast Abstract Interpretation with Tristate Numbers. In *Proceedings of CGO*, page 254–265. IEEE Press, 2022.

[45] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. Verifying the Verifier: eBPF Range Analysis Verification. In Constantin Enea and Akash Lal, editors, *CAV*, pages 226–251, 2023.

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

[47] Dmitry Vyukov and Andrey Konovalov. Syzkaller: an Unsupervised Coverage-guided Kernel Fuzzer. https://github.com/google/syzkaller, 2015.

[48] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful assert statements for unit test cases. In *Proceedings of ICSE*, page 1398–1409, New York, NY, USA, 2020.

[49] Jingyue Wu, Gang Hu, Yang Tang, and Junfeng Yang. Effective Dynamic Detection of Alias Analysis Errors. In *Proceedings of ESEC/FSE*, page 279–289, 2013.

[50] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of PLDI*, page 283–294, 2011.

[51] Hao Yu, Yiling Lou, Ke Sun, Dezhi Ran, Tao Xie, Dan Hao, Ying Li, Ge Li, and Qianxiang Wang. Automated assertion generation via information retrieval and its integration with deep learning. In *Proceedings of ICSE*, page 163–174, New York, NY, USA, 2022.

[52] Shung-Hsi Yu. Model Checking (a very small part) of BPF Verifer. https://gist.github.com/shunghsiyu/a63e08e6231553d1abdece4aef29f70e. Accessed: Nov 20, 2023.

[53] Chengyu Zhang, Ting Su, Yichen Yan, Fuyuan Zhang, Geguang Pu, and Zhendong Su. Finding and Understanding Bugs in Software Model Checkers. In *Proceedings of ESEC/FSE*, page 763–773, 2019.

[54] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. XRP: In-Kernel Storage Functions with eBPF. In *OSDI*, pages 375–393, July 2022.