Approximation Enforced Execution of Untrusted Linux Kernel Extensions

Hao Sun
ETH Zurich

Zhendong Su
ETH Zurich

Abstract

Modern OS kernels allow untrusted extensions, such as eBPF programs, to be dynamically loaded into kernel space, with their safety ensured by an *in-kernel verifier*. However, this approach implicitly places the entire verifier, a complicated and error-prone component, within the trusted code base. Despite substantial efforts to verify and test the verifier, its complexity and frequent updates continue to introduce soundness bugs, leading to various security issues.

This paper introduces *Approximation-Enforced Execution (AEE)*, a novel concept to ensure the safe execution of untrusted kernel extensions, even in the *presence* of potential verifier bugs. The verifier can be essentially abstracted into two key components: the complex state approximation and the simpler safety check based on the former. By enforcing the program execution to remain within the verifier's approximations, the soundness of state approximation is, by design, not assumed—executions with non-contained states are terminated, thereby significantly reducing the trust base. AEE also leverages the verifier, but mainly obtains the approximations. It then rewrites the program to conduct the approximation enforcement, where trust is established by combining the runtime facts with minimal reliance on the verifier's safety checks. We apply AEE to ensure the spatial memory safety of eBPF programs and formally prove its soundness *w.r.t.* mitigating the verifier's soundness bugs and completeness *w.r.t.* ensuring safety under the reduced trust base. Our evaluation shows that our prototype reduces the trusted code base by *4.5x*, with an average runtime overhead of 1.2% and an average increase in binary size of 4.8%.

1 Introduction

Modern operating system (OS) kernels are designed to be highly extensible to support diverse user space workloads. Various kernel functionalities, such as file system [42, 100] and network management [69], can be dynamically tailored through loadable extensions. A prominent mechanism facilitating such extensions is eBPF [18], which enables user space to augment the Linux kernel with customized bytecode extensions and supports a wide range of applications, including packet processing [89], security monitoring [38, 57, 59], and performance profiling [30, 63], among others [48, 53, 62].

Because these extensions are executed in privileged mode and developed by third parties, they are inherently untrusted and potentially malicious (see Section 2 for details on the threat model). Accordingly, various approaches have been proposed to ensure the safe execution of these extensions, such as fault isolation [67]. To balance safety, flexibility, and performance, modern kernels increasingly rely on automated verification [51]. In eBPF, this is implemented with an *in-kernel verifier* [24] that employs abstract interpretation [46] to analyze each eBPF program systematically and checks safety properties (*e.g.*, memory safety) before allowing its execution. As illustrated in Figure 1, the verifier serves as a fundamental gatekeeper, establishing trust between untrusted extensions and the kernel.

Figure 1: The existing workflow of loading extensions.

The Hidden (Unreliable) Trust. The verifier determines whether a program is safe by mapping programs to either *safe* or *unsafe*, formally expressed as $V : \mathcal{P} \to \{safe, unsafe\}$, where $\mathcal{P}$ is the set of all possible programs. Conventionally, a program $P \in \mathcal{P}$ is deemed safe for execution if it passes V's verification, captured by:

$$V(P) = \text{safe} \implies Safe(P) \tag{1}$$

However, this view overlooks a crucial assumption: the predicate $Safe(P)$ relies not only on V designating the program as safe but also on the soundness of the verifier itself. Hence, Equation 1 is more accurately framed as:

$$V(P) = \text{safe} \wedge \textbf{Sound}(V) \implies Safe(P) \tag{2}$$

In practice, the verifier's soundness, *i.e.*, $Sound(V)$, is often taken for granted.

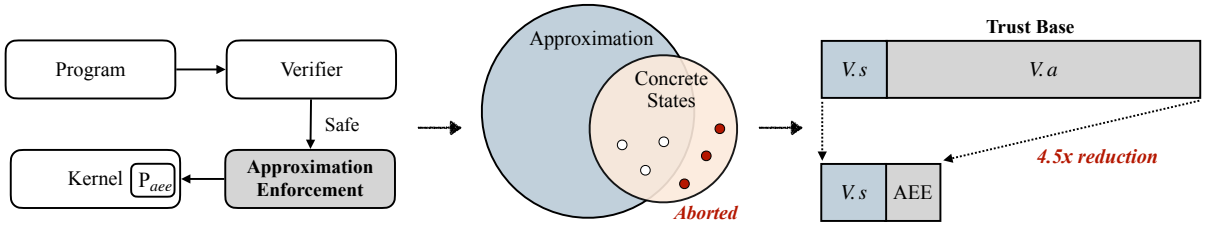

Figure 2: Workflow of AEE. Programs that pass the verifier are transformed to perform approximation enforcement. Therefore, $V.a$ is untrusted and executions that deviate from the expected state are terminated, achieving a 4.5x trust base reduction.

To ensure safe execution, the verifier conducts two essential steps, each requiring correctness for overall soundness. First, it collects the program state approximation according to instruction semantics, *i.e.*, *approximation transition*, denoted as $V.a$. The approximation must soundly capture all possible concrete states using certain abstract domains, *e.g.*, all possible scalar values of a variable can be captured in the interval domain [47]. Second, it applies *safety checks* ($V.s$) using that approximation to determine whether each instruction is safe. Thus, Equation 2 can be expanded into:

$$V(P) = \text{safe} \wedge \mathit{Sound}(V.s) \wedge \mathit{Sound}(V.a) \implies \mathit{Safe}(P) \quad (3)$$

where $\mathit{Sound}(V.a)$ and $\mathit{Sound}(V.s)$ denote the soundness of $V.a$ and $V.s$, respectively.

In practice, $V.s$ is relatively straightforward: it involves checking whether certain properties (*e.g.*, pointer offsets during memory accesses) remain within safe bounds determined by the approximation. In contrast, correctly approximating program states ($V.a$) is significantly more challenging. It must account for all possible concrete states across diverse abstract domains. For example, the eBPF verifier tracks variable ranges in five separate abstract domains [71, 77, 91] and models the stack at a byte-level granularity. Our evaluation reveals that $V.a$ constitutes over 83% of the verifier's implementation logic, which has led to numerous soundness bugs [8, 9, 35, 36, 72]. ***An unsound approximation can omit certain reachable concrete states***, allowing attackers to construct malicious programs that exploit such states (detailed in Section 3). Although the verifier may deem such programs safe using the unsound approximation, their execution on states omitted from the approximation can introduce severe vulnerabilities, such as local privilege escalation [10–12, 14, 15]. Consequently, $V.a$ represents a substantial yet potentially unreliable component of the trust base.

Substantial efforts have been made to improve the verifier's approximation transitions ($V.a$) given the importance of $\mathit{Safe}(P)$. Researchers have applied formal verification to ensure the soundness of specific components, such as the `tnum` abstract domain [41, 75, 77, 91, 92, 97], and have developed testing approaches that utilize effective testing oracles for validating the eBPF verifier [23, 43, 64, 85]. Despite these advances, soundness bugs in $V.a$ persist, often arising from discrepancies between the verified specifications and the actual implementations [17, 26]. Moreover, because the eBPF verifier is continually updated to accommodate more complex kernel extensions, the emergence of new soundness bugs is unavoidable [2, 3, 6, 16, 28, 37]. This predicament raises a fundamental question: if the verifier is the gatekeeper of untrusted extensions, then *who will guard the guard*? Or, in the Latin phrase:

Quis custodiet ipsos custodes?

New Perspective. In this paper, we propose to ensure $\mathit{Safe}(P)$ by *reframing* how the verifier's trustworthiness is viewed. Unlike prior work that aims to confirm $\mathit{Sound}(V)$ in its entirety, we *relax* this requirement by explicitly minimizing the hidden and unreliable trust placed in the verifier—particularly in its approximation logic. Our approach strategically redefines which components to trust and untrust, preserving kernel integrity even in the presence of verifier flaws.

Focusing on the verifier's two core components, we observe that: (1) while $V.a$ is inherently complex, the safety checks $V.s$ are comparatively simple and account for a much smaller fraction of the verifier's codebase, and (2) enforcing approximations at runtime is significantly simpler than constructing them statically. This leads to our key insight: if runtime execution is enforced to remain within the verifier's statically approximated state space, then soundness of the approximation logic $V.a$ is no longer required and *untrusted*. Instead, any deviation from the expected state at runtime can be detected and terminated, and other permitted executions must align with the approximated state, with their safety ensured by the *trusted* static check $V.s$, a much smaller trust base. In other words, we decouple trust from the approximation logic $V.a$ by enforcing its outcome at runtime. While the verifier may still produce unsound approximations, such flaws no longer compromise kernel safety: ***any unsafe state omitted by unsound approximations will trigger the enforcement that halts the execution***. We distill this approach into the following statement, which modifies Equation 3:

$$V(P) = \text{safe} \wedge \mathbf{Sound}(V.s) \implies \mathit{Safe}(P_{aee}) \quad (4)$$

where $\mathit{Safe}(P_{aee})$ denotes the safety under our enforcement.

Our Approach. Building on our key insight, we propose *Approximation-Enforced Execution* (AEE), a novel approach to ensuring the safe execution of untrusted kernel extensions.

As depicted in Figure 2, AEE transforms a verifier-approved program P into an instrumented variant P_{aee} by inserting assertions. These assertions enforce that the program's concrete execution states remain within the verifier's statically computed approximations; otherwise, execution is terminated. When the verifier's approximations are sound, P_{aee} proceeds unimpeded; in cases where unsound approximations arise, immediate termination on executions with omitted states prevents any adverse impact on the kernel. Since the approximations are enforced, $Safe(P_{aee})$ holds provided the associated safety checks are correct. From a threat mitigation perspective, malicious adversaries *attack* the verifier by exploiting states overlooked by unsound approximations; AEE *mitigates* such attacks by enforcing strict adherence to the approximations.

We applied AEE to enforce spatial memory safety of eBPF extensions by extending the verifier (Section 7 details AEE's generality). We begin by formalizing eBPF in a small core language, which allows us to rigorously (1) identify the set of operations requiring enforcement, and (2) express AEE's semantics within the formal framework. Next, we prove AEE's soundness *w.r.t.* mitigating unsound approximations (*i.e.*, every termination must indicate a verifier's bug) and prove its completeness *w.r.t.* safety assurance (*i.e.*, permitted execution must be safe) under the reduced trust base. Our evaluation demonstrates that AEE protects the kernel from a wide range of malicious eBPF programs exploiting verifier soundness bugs identified over the past five years. By adopting AEE, we observe a *4.5x* reduction in the trusted code base (*i.e.*, bugs in those components can no longer compromise the kernel), at the cost of 4.8% in binary size and 1.2% in runtime overhead. Our key contributions are as follows:

- We introduce Approximation-Enforced Execution, a novel mechanism to ensure the safe execution of untrusted kernel extensions while substantially reducing the trust base and mitigating security risks.
- We apply AEE to eBPF for spatial memory safety by identifying relevant operations and designing enforcement under a formal framework.
- Through a comprehensive experimental study, we show that AEE provides a much-improved safety guarantee while incurring minimal overhead.

2 Threat Model

The eBPF subsystem facilitates the dynamic loading of kernel extensions, allowing fine-grained control and system behavior monitoring without requiring full kernel modifications. Under typical usage, developers write eBPF extensions and submit the bytecode to the kernel via a dedicated system call. Before the program is loaded, the eBPF verifier analyzes the code to ensure it compiles to the safety requirements, such as bounded loops and valid memory accesses. Upon successful verification, the kernel places the program at a designated hook point, where it executes in the same address space as the kernel and can observe and manipulate specific events.

The primary concern is that attackers may exploit verifier flaws to install malicious kernel extensions. In particular, we focus on logic bugs in the verifier's state tracking—namely, unsound approximations—that enable bypassing or confusing the verifier's checks. We assume that attackers can craft and submit eBPF bytecode and aim to exploit the verifier's flaw to achieve privileged operations such as arbitrary memory accesses. Although some Linux distributions disable unprivileged eBPF, requiring capabilities such as `CAP_BPF` [45, 83], our assumption remains justified for two primary reasons. First, real-world deployments commonly enable eBPF in server or container environments for diverse tasks [38, 52, 59]. Second, from the perspective of kernel maintainers, the user-space configurations are not guaranteed; any bug in the verifier may still allow erroneous programs to be loaded, compromising kernel integrity. Consequently, verifier flaws continue to pose a significant threat to modern systems.

3 Illustrative Example

This section describes how the verifier blocks unsafe extensions, how adversaries exploit its unsound approximations, and how our approach, AEE, effectively mitigates these risks.

The Verifier. The eBPF verifier utilizes abstract interpretation to analyze each extension. It examines all feasible control-flow paths and aggregates potential program states at each program point, representing them as approximations ($V.a$). A program is considered safe if, for every approximation, the corresponding operation adheres to specific properties ($V.s$). The verifier maintains soundness under two conditions: (1) each approximation encompasses all possible runtime values a variable may take, and (2) the associated safety checks on these states are correct. Conversely, if the approximations exclude reachable states, the verifier fails to validate executions with them, thereby permitting potentially unsafe behaviors.

Figure 3 illustrates an unsafe extension rejected by the verifier based on sound approximation. At instruction #2, a logical AND constrains the value of register `r2`. The verifier concludes that `r2` may only hold $\{0, 1\}$, a sound approximation encompassing all runtime values. It then updates the approximation according to the semantics of each instruction, as part of $V.a$. At #4, the program attempts to load data from a user-accessible region, with the access size determined by `r2`. Given that the verifier has inferred, through its approximation, that `r2` could be an out-of-bounds size (*e.g.*, 255), it rejects the program via the corresponding safety check $V.s$. Although Figure 3 presents a simplified scenario, real-world eBPF programs are considerably more complex, encompassing intricate operations and diverse state transitions. Accurately validating such behaviors requires faithful modeling of

```
1: r1 = fp-16          ; 16 bytes area
2: r2 &= 1             ; r2 ∈ [0, 1]
3: r2 += 254           ; r2 ∈ [254, 255]
4: load(uptr, r1, r2) ; unsafe, statically rejected
```

Figure 3: An unsafe program correctly rejected by the verifier. `fp-16` refers to the stack pointer with a 16-byte offset. `load()` retrieves data from user-accessible regions (*e.g.*, `bpf_map`), denoted as `uptr` for simplicity.

instruction semantics. Any flaw in this approximation logic risks overlooking unsafe states, motivating the design of AEE.

Attack. In practice, *V.a* can become unsound, leading to states that are *reachable at runtime* but remain *unaccounted for* in the approximation. Such omissions compromise the validity of the verifier's safety assessments. Figure 4(a) illustrates this scenario. At #4, `r2`, with a tracked range of $[254, 255]$, is cast to a signed 8-bit integer (`s8`) and then sign-extended to 64 bits. Correctly, this operation should yield a 64-bit range of `[u64_max-1,u64_max]`, as the values in `r2`'s range have their sign bits set. However, due to a flaw in the verifier's approximation logic [37], the higher 32 bits are mistakenly zeroed, resulting in an unsound approximation: `[u32_max-1,u32_max]`. This oversight excludes reachable runtime values, *e.g.*, `u64_max`, from the verifier's purview. Consequently, these omitted values escape the verifier's safety checks, allowing executions associated with them to bypass static validation and lead to safety violations

We refer to any execution path involving such omitted states as an *escaped execution*, and paths with runtime states contained within the approximations as *checked executions*.

Crucially, unsound approximations can be potentially exploited by attackers. First, they craft an instruction sequence that triggers the soundness bug, thereby creating an escaped execution. Second, they carefully design the remainder of the program such that the verifier still deems it safe, while the omitted state at runtime enables malicious behavior. In Figure 4(b), after the faulty sign-extension, the verifier believes that `r2` has zero in its higher 32 bits. At runtime, however, those bits are set to all ones. The verifier thus concludes *incorrectly* that a subsequent right-shift operation yields a constant zero, when in fact it produces one. Subsequently, `r2` can be used to control a pointer on the stack: the verifier assumes the memory operation is safe according to its unsound approximation, yet in reality, the spilled pointer is overwritten, allowing an attacker to corrupt memory and escalate privileges. Although a complete exploit may involve more operations to confuse the verifier, the core principle persists: manipulate non-contained states in the escaped execution while preserving a superficially "safe" approximation for the verifier.

AEE as Mitigation. Since accurately approximating program states involves a large and potentially unreliable codebase, AEE aims to exclude it from the trust base. The core idea is that attackers exploit the flaw by manipulating non-contained

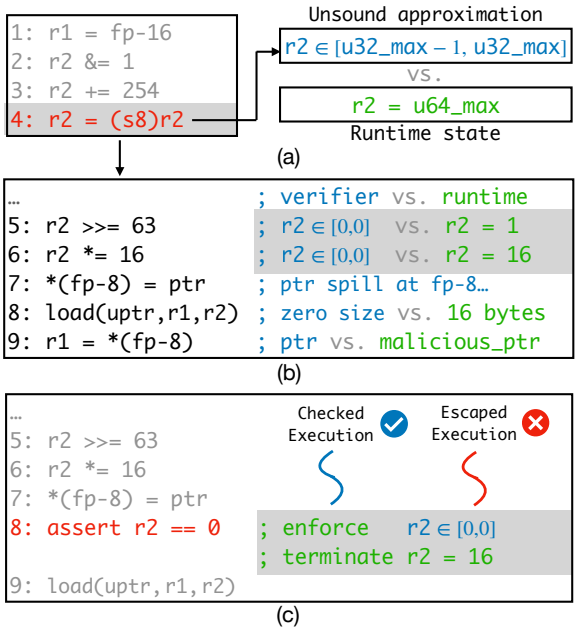

Figure 4: Exploit program and AEE's enforcement. Comments after each semicolon show the verifier's approximation (blue) and the actual runtime state (green).

states; mitigation, in turn, involves *permitting* only checked executions and *terminating* escaped ones.

Concretely, once a program passes the verifier, rather than allowing immediate execution in the kernel, AEE transforms the program by inserting assertions that strictly enforce the runtime states to remain within the approximation—regardless of whether those approximations are sound. When an (escaped) execution diverges from the approximation, it is terminated as its safety is unknown. Figure 4(c) illustrates this process. AEE injects an assertion before using `r2` as a memory access size parameter. Since the verifier checks access safety assuming `r2=0` (the sole state encompassed by the unsound approximation), the assertion permits only this *checked execution*. Consequently, although the exploit program may be loaded, it is immediately terminated during any *escaped execution* where `r2=16` (a state omitted from static checks), effectively thwarting the attack. Under AEE, only checked executions with contained states are permitted. Thus, execution remains safe, provided the safety checks on the enforced approximation are correct. The complex approximation logic is no longer trusted—its outputs are enforced, significantly reducing the trust base.

Applying AEE to enforce specific properties poses ***two main challenges***: (1) identifying operations that require enforcement and (2) determining the property each enforcement must maintain. We address these by formalizing eBPF's core semantics, enabling systematic identification of enforcement points and rigorous reasoning about enforcement effects. The formalization also decouples AEE from specific implementations, allowing its adoption by any conforming language.

$$
\begin{aligned}
cmd &::= w := E \mid w :=_{sz} *p \mid *p :=_{sz} x \mid w := \texttt{func}(x,y) \\
&\quad \mid \texttt{assume}(B) \mid w := \texttt{mem}\ K \\
E &::= K \mid x \mid x+y \mid x-y \\
B &::= x = y \mid x \neq y \mid x \leq y
\end{aligned}
$$

Figure 5: The syntax of the core eBPF language. `K` is a constant, `sz` can only be one, two, four, or eight bytes, and `func` represents a function call.

$$
\begin{aligned}
t &\in \text{Tag} = \mathcal{T}_{mem} \sqcup \{T_s\} \\
v &\in \text{Value} = \text{Tag} \times \mathbb{Z} \\
a &\in \text{Address} = \mathcal{T}_{mem} \times \mathbb{Z} \\
c &\in \text{Cell} = \text{Address} \times \text{Size} \\
e &\in \text{Env} = \text{Register} \to \text{Value} \\
\mu &\in \text{Mem} = \text{Cell} \to \text{Value} \\
\sigma &\in \text{State} = \text{Env} \times \text{Mem}
\end{aligned}
$$

Figure 6: Semantic domains. $\mathcal{T}_{mem}$ represents an unbounded set, and each element in it is a unique memory tag, and T_s is a tag shared by all scalar values.

4 The eBPF Language

This section formalizes eBPF into a small core language, which captures the essence of eBPF programs while abstracting away certain details. To maintain consistency, we follow most notations adopted by Gershuni *et al.* [51] whenever possible. However, their formalization aligns with the verifier's analysis algorithm. For instance, concepts such as shared and private memory are not reflected in program execution. We adapt the formalization to more closely match the semantics of real execution and extend it to support the `call` instruction.

Syntax. Figure 5 illustrates the syntax of the core language. An eBPF program is represented as a control graph, with each edge annotated by a command. The symbols `x`, `y`, `w`, and `p` represent meta-variables. Primitive commands conduct arithmetic operations, access memory in sizes of one, two, four, or eight bytes, and invoke a set of predefined helper functions. We abstract the function call into a form that takes two parameters, sufficient to represent both basic and memory-accessing calls, in which case the memory address and access size are provided in `x` and `y`, respectively. The `assume` command filters states not satisfying the predicate `B`, serving as an abstraction for the conditional jump. Programs obtain pointers statically via the `mem` command, which associates a memory region of known size `K` with `w`.

Machine State. Figure 6 defines the machine state, which consists of a tuple $\sigma = (e, \mu)$, representing the environment and memory state. e maps a register to its value and μ map a memory cell to its value. Programs use a fixed set of registers, $\text{Register} = \{r0, \ldots, r10\}$, ranged over by the meta-variables. The machine is stuck when evaluated outside the domain.

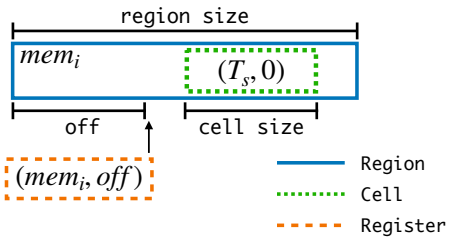

Figure 7: An illustrative example of register, cell, and region.

Registers and memory cells store *tagged* values, represented as (t, n), where $n \in \mathbb{Z}$ is the actual value, and $t \in \text{Tag}$ is a tag that differentiates the semantics of the value. The function $e()$ takes a register name and returns its value. Additionally, $e_n()$ and $e_t()$ are helper functions that extract the value and tag part of a register, respectively. Values are categorized into two types: scalar and address (pointer). The scalar value is (T_s, n), where T_s and n represents the scalar tag and the value, respectively. An address value is tagged with a unique identifier of the memory region they point to, from the unbounded tag set $\mathcal{T}_{mem}$, and n represents the offset within the memory region.

As shown in Figure 7, each program consists of a set of memory regions, which are disjoint, contiguous, and byte-addressable, with each region having a unique identifier and a known size. The helper function $sizeof()$ takes a memory tag and returns its byte size. A memory cell is defined by an address and a size value, where the address specifies the region and offset, and the latter represents the size of the cell in bytes. The size is constrained to one, two, four, or eight bytes, corresponding to valid memory access sizes. The function $\mu()$ retrieves the value of a cell, and the actual value and tag of a cell c are denoted as $\mu_n(c)$ and $\mu_t(c)$, respectively.

Both `r1` and `r10` store an address value in the initial machine state. Additionally, $\overline{c} = \{(\mu_t(c), i) \mid \mu_n(c) \leq i < \mu_n(c) + sz\}$ denotes the set of addresses within a memory cell c of size sz, which is used to determine whether two cells c_i and c_j overlap, *i.e.*,, $\overline{c_i} \cap \overline{c_j} \neq \emptyset$ indicates an overlap.

Operational Semantics. Figure 8 presents the effect of each command on the machine state, expressed in the style of small-step operational semantics. Intuitively, a program performs arithmetic operations between registers, stores and loads values to and from memory cells, invokes a limited set of functions, and follows control flow based on the filter conditions.

The assignment commands associate an immediate value or the value from the source register with the destination register, while the memory state μ remains unchanged (Equations 5-6). The operands of arithmetic can be either a scalar or an address value. Arithmetic between two address values results in an arbitrary scalar value, which is omitted from the rule.

$$\langle w := K, \sigma \rangle \Rightarrow (e[w \mapsto (T_s, K)], \mu) \tag{5}$$

$$\langle w := x, \sigma \rangle \Rightarrow (e[w \mapsto e(x)], \mu) \tag{6}$$

$$\langle w := x + y, \sigma \rangle \Rightarrow (e[w \mapsto (e_t(x), e_n(x) + e_n(y))], \mu) \tag{7}$$

$$\langle w := x - y, \sigma \rangle \Rightarrow (e[w \mapsto (e_t(x), e_n(x) - e_n(y))], \mu) \tag{8}$$

$$\langle w := \texttt{mem}\ \ K, \sigma \rangle \Rightarrow (e[w \mapsto (t, 0)], \mu) \quad \text{where } t \in \mathcal{T}_{mem} \wedge sizeof(t) = K \tag{9}$$

$$\langle w := \texttt{func}(x, y), \sigma \rangle \Rightarrow (e[w \mapsto v, x \mapsto \bot, y \mapsto \bot], \mu') \quad \text{where } (v, \mu') = eval(\texttt{func}, e(x), e(y), \mu) \tag{10}$$

$$\langle *p :=_{sz} x, \sigma \rangle \Rightarrow (e, \mu') \quad \text{where } \mu' = Store(\mu, (e(p), sz), e(x)) \tag{11}$$

$$\langle w :=_{sz}\ *p, \sigma \rangle \Rightarrow (e[w \mapsto \mu(c)], \mu) \quad \text{where } c = (e(p), sz) \tag{12}$$

$$\langle \text{assume}(x = y), \sigma \rangle \Rightarrow \sigma \qquad \text{if } e(x) = e(y) \tag{13}$$

$$\langle \text{assume}(x \neq y), \sigma \rangle \Rightarrow \sigma \qquad \text{if } e(x) \neq e(y) \tag{14}$$

$$\langle \text{assume}(x \leq y), \sigma \rangle \Rightarrow \sigma \qquad \text{if } e_n(x) \leq e_n(y) \tag{15}$$

Figure 8: Operational semantics. Each rule defines a state transition in the form of $\langle cmd, \sigma \rangle \Rightarrow \sigma'$.

$$Store(\mu, c, (t, n)) = \mu'[c \mapsto (t, n)] \quad \text{where } \mu' = \mu[c_0 \mapsto \bot \mid c_0 \in dom(\mu) \wedge \overline{c_0} \cap \overline{c} \neq \emptyset]$$

Figure 9: The *Store*() helper sets the cell to the value and resets all the overlapped cells.

For operations involving a scalar and an address value, we constrain the address to be the left operand. Consequently, the arithmetic command sets the tag of the destination register as the left operand, *i.e.*, $e_t(x)$, and the value is computed using the values of both operands (Equations 7-8). The `mem` command obtains a memory region with tag $t \in \mathcal{T}_{mem}$ and size K, storing the address value $(t, 0)$ in the register (Equation 9). The `func` command invokes a function using the input registers and stores the result in the destination register. The helper function *eval*() evaluates the function with the parameters and the memory state, and returns both the result and the updated memory state. The source registers are set to arbitrary scalars to simulate the side effects of the call (Equation 10). When `x` contains an address, `y` represents the memory access size.

The semantics of the store command are defined using the *Store*() helper (Equation 11). *Store*() takes the memory state, the target cell c, and the value, updates c with the value, and resets all overlapping cells (Figure 9). The load command retrieves the value associated with a cell and returns an arbitrary scalar if the cell is not in μ (Equation 12). The `assume` command filters out states that do not satisfy the given predicate, without side effects on the satisfying states. The comparisons are restricted to values with the same tag (Equations 13-15).

5 Approximation Enforced Execution

This section describes and contrasts AEE with the existing approach and its application to ensure memory safety.

5.1 The Essence of AEE

The execution of an eBPF program can be abstracted as a state transition on each command $\langle cmd, \sigma \rangle \Rightarrow \sigma'$ as defined in Section 4. To ensure the safe execution of untrusted kernel extensions, we adopt the following notation to represent the safety condition that must be held at each command:

$$Safe(cmd, \sigma) \tag{16}$$

For instance, the safety condition for memory access requires that the operand must be an address value and the access must be within bounds. The core language, consistent with eBPF's design, does not enforce such conditions in its operational semantics; instead, the safety is provided by the verifier.

The Verifier. eBPF adopts an abstract interpretation-based verifier as a gatekeeper, *i.e.*, programs passing it are deemed safe for execution. The verifier ensures the safety conditions by collecting the state approximation at every program location, denoted as $\sigma^{\#}$, and adapts the semantics shown in Figure 8 to conduct the safety check on the approximation, represented as $Safe^{\#}(cmd, \sigma^{\#})$, before every transition:

$$\langle cmd, \sigma^{\#} \rangle \Rightarrow_{\#} \begin{cases} \sigma^{\#\prime} & Safe^{\#}(cmd, \sigma^{\#}) \\ \bot & \text{otherwise} \end{cases} \tag{17}$$

Take $w :=_{sz} *p$ for instance. The verifier approximates the range of the offset of p and verifies if the access specified by the range is within bounds; otherwise, the verification fails. Next, it interprets the command in the corresponding abstract domains and transfers $\sigma^{\#}$ to $\sigma^{\#\prime}$, simulating the command.

With this approach, $Safe(cmd, \sigma)$ holds only if (1) every concrete state is contained in the corresponding approximation, and (2) the safety check conducted on the approximation implies that on the concrete states. Therefore, $Sound(V.a) \wedge Sound(V.s)$ in Equation 3 can be further reformulated as Equation 18, where $\gamma(\sigma^{\#})$ represents the set of concrete states contained in $\sigma^{\#}$:

$$\forall \sigma \in \gamma(\sigma^{\#})\ \ (\langle cmd, \sigma \rangle \Rightarrow \sigma' \wedge \langle cmd, \sigma^{\#} \rangle \Rightarrow_{\#} \sigma^{\#\prime}) \implies \sigma' \in \gamma(\sigma^{\#\prime}) \wedge Safe(cmd, \sigma) \tag{18}$$

Ideally, since the verifier traverses each potential path and each command is checked against the over-approximation, the program is safe once it passes the verification. In practice, soundness bugs continue to surface in the state approximation procedure due to its inherent complexity and frequent changes and improvements, making it unreliable.

Our Approach. AEE is a novel concept to provide strong safety guarantees for the execution of untrusted extensions. First, rather than using the verifier as a gatekeeper (thus trusting it can correctly approximate all the concrete states), AEE views the verifier as a safety statement generator. More concretely, for a program $P \in \mathcal{P}$, AEE first uses the verifier for verification, yet $V(P) = \text{safe}$ does not imply the program is safe, but produces a safety statement for each command:

$$\forall \sigma \in \gamma(\sigma^{\#})\ \ Safe^{\#}(cmd, \sigma^{\#}) \implies Safe(cmd, \sigma) \tag{19}$$

From the perspective of AEE, the verifier does not claim all possible concrete states are contained but tells that each concrete state σ that is *already approximated* in $\sigma^{\#}$ is checked to be safe for execution, and more importantly, there could exist a reachable concrete state *not contained*, *i.e.*, $\exists \sigma \notin \gamma(\sigma^{\#})$, and the safety with non-contained states is undetermined.

Therefore, after passing the verifier, AEE further transforms the program P to P_{aee} to enforce the concrete states during the execution to remain within the approximation, *i.e.*, non-contained states result in termination. AEE essentially adapts the semantics with a containment check:

$$\langle cmd, \sigma \rangle \Rightarrow_{aee} \begin{cases} \sigma' & \sigma \in \gamma(\sigma^{\#}) \wedge \langle cmd, \sigma \rangle \Rightarrow \sigma' \\ \bot & \text{otherwise} \end{cases} \tag{20}$$

Figure 10 illustrates AEE's effects on all the verifier-approved programs, where four cases could happen:

- #1: ***Safe*** programs, accepted by ***sound approximations***, execute normally as every reachable state is permitted.
- #2: ***Safe*** programs, accidentally accepted by ***unsound approximations***, are terminated when omitted states occur, a rare case explicitly indicating a soundness bug.
- #3: ***Unsafe*** programs, accepted due to ***incorrect safety checks*** ($V.s$), are *not aborted*, a rare case violating the assumption of AEE.
- #4: ***Unsafe*** programs, incorrectly accepted due to ***unsound approximations*** (thus executed with the existing approach), are terminated by AEE once the non-contained states are encountered.

Section 7.3 illustrates #2 and #3 with concrete examples. AEE only terminates a program in cases #2 and #4 due to unsound approximation. Hence, AEE is sound *w.r.t.* mitigating soundness bugs, *i.e.*, termination must indicate verifier flaws.

By aborting escaped executions with non-contained states, the effects of potential unsound approximations are effectively eliminated, and the kernel remains unaffected even in the presence of potential bugs. With AEE, $Safe(cmd, \sigma)$ holds as long as the safety check conducted on the approximation implies it, *i.e.*, Equation 19. Since the safety check is much more straightforward and involves a smaller code base than the state approximation, AEE provides improved safety with a significantly reduced trust base.

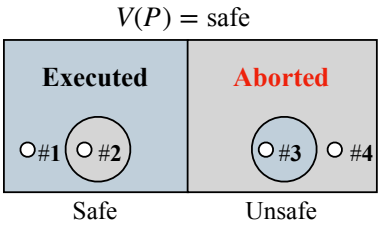

Figure 10: AEE's effects on the verifier-approved programs ($V(P) = \text{safe}$). Programs represented by #1 (safe) and #3 (unsafe) in the blue area are executed, while #2 (safe) and #4 (unsafe) in the grey area are aborted.

5.2 Applying AEE for Memory Safety

Unsound approximations can arise from various components, *e.g.*, range analysis and path pruning. Regardless of origin, attackers exploit these flaws ultimately to overwrite kernel memory, *e.g.*, credential structures. By enforcing that extensions access only permitted regions, we significantly restrict their impact on the kernel. Therefore, rather than analyzing each verifier component, AEE enforces memory safety as a sink point for soundness bugs—blocking malicious accesses eventually instead of considering individual bugs. We focus on spatial memory safety, as temporal issues are limited in eBPF: unprivileged users cannot perform dynamic allocations. Applying AEE involves *three steps*: (1) define the safety conditions for related commands, (2) determine the safety statements produced by the verifier for those commands, and finally (3) enforce the approximations based on the statements and demonstrate the safety conditions defined in (1) hold.

Memory Safety. We now define the meaning of memory safety for the core language by first identifying the relevant commands that modify or access the memory state. According to the operational semantics shown in Figure 8, only the memory load, store, and helper call commands may alter or access the memory state μ, *i.e.*, the state keeps the same and is not visited before and after the transitions of other commands. For instance, the arithmetic and conditional commands operate on the registers only, and μ does not change in Equations 5-9 and Equations 13-15. Therefore, the memory safety depends on the safe execution of the $*p :=_{sz} x$, $w :=_{sz} *p$, and $w := \texttt{func}(x, y)$ commands.

We now define the safety conditions for those commands. We use $InBounds()$ to denote the condition that memory access must be in bounds, and the function takes an address value (t, n), and access size sz as inputs:

$$InBounds((t, n), sz) \ = \ 0 \leq n \leq sizeof(t) - sz \tag{21}$$

For memory load ($w :=_{sz} *p$), and store ($*p :=_{sz} x$), the safety conditions are: (1) the register contains an address value, and (2) the access is within the bounds, denoted as:

$$e_t(p) \in \mathcal{T}_{mem} \wedge InBounds(e(p), sz) \tag{22}$$

For $w := \text{func}(x, y)$ that conducts memory access, the first register x consists of the address value, and y represents the access size. Therefore, the safety conditions, in this case, are: (1) the value of x is an address, and (2) the potential accessed memory range by the function must be in bounds, denoted as:

$$e_t(x) \in \mathcal{T}_{mem} \wedge \mathit{InBounds}(e(x), e_n(y)) \tag{23}$$

Equation 22 in conjunction with Equation 23 describes the meaning of memory safety for the core language.

Safety Statements. The safety statements obtained from the verifier essentially state under what approximations the command is safe to execute. An approximation is an abstract value derived by the verifier within an abstract domain, which is

$$v^{\#} \in \text{Value}^{\#} = \text{Tag} \times \mathbb{Z} \times \mathbb{Z}$$

An abstract value is also a tagged value and consists of two parts: (1) the tag, which is from the same domain Tag as concrete values, (2) the value range $(lo, hi) \in \mathbb{Z} \times \mathbb{Z}$. The value range represents the scalar range if the tag is T_s; otherwise, it represents the range of the pointer offset. We adopt the same set of notations as in the concrete domain with the # mark to denote the corresponding notations in the abstract domain, and additionally, we use $e_l^{\#}()$ to $e_h^{\#}()$ to obtain the lower and higher bound of the verifier's range, respectively.

From the perspective of AEE, the safety statement produced by the verifier for the memory load and store commands $w :=_{sz} *p$, $*p :=_{sz} x$ is:

$$\begin{aligned} e_t(p) = e_t^{\#}(p) \in \mathcal{T}_{mem} \wedge e_l^{\#}(p) \leq e_n(p) \leq e_h^{\#}(p) \\ \wedge \mathit{InBounds}^{\#}(e^{\#}(p), sz) \implies \mathit{InBounds}(e(p), sz) \end{aligned} \tag{24}$$

The first two predicates correspond to $\forall \sigma \in \gamma(\sigma^{\#})$ in Equation 19, and $\mathit{InBounds}^{\#}()$ function corresponds to the safety check $\mathit{Safe}^{\#}()$ conducted by the verifier for memory accesses. The statement indicates that for an address value p, whose tag is the same as the verifier's tracked tag, and offset is between the collected lower and higher bounds, the memory access with sz bytes is checked to be safe, *i.e.*, $\mathit{InBounds}^{\#}()$ implies $\mathit{InBounds}()$ under this presumption.

When the first register is an address, the safety statement produced for the call command $w := \text{func}(x, y)$ is:

$$\begin{aligned} e_l^{\#}(y) \leq e_n(y) \leq e_h^{\#}(y) \\ \wedge\ e_t(x) = e_t^{\#}(x) \in \mathcal{T}_{mem} \wedge e_l^{\#}(x) \leq e_n(x) \leq e_h^{\#}(x) \\ \wedge\ \mathit{InBounds}^{\#}(e^{\#}(x), e_n^{\#}(y)) \implies \mathit{InBounds}(e(x), e_n(y)) \end{aligned} \tag{25}$$

Equation 25 states that, for a function call that accesses the memory pointed by x, where the tag of x matches the verifier's tracked tag and the offset of x and the value of y are between the collected bounds, the access is checked to be safe.

Enforcing Approximation. With the statements produced, AEE transforms the program P to P_{aee} with assertions to enforce the concrete states to remain within the approximations, thereby ensuring the predicates of statements always hold. We describe P_{aee} by defining its operational semantics based on the semantics of P, and we show how each transition rule ensures those predicates hold.

Offset Enforcement. First, we ensure, for an address p, the predicate $e_l^{\#}(p) \leq e_n(p) \leq e_h^{\#}(p)$ holds, *i.e.*, enforcing the pointer offset to remain within the verifier's tracked range. Conceptually, one can assert the offset $e_n(p)$ before its usage, which in this context includes the memory load, store, and call commands. However, a pointer's offset needs to be computed by subtracting the runtime pointer value from its base address. Since the base address of a pointer is unknown during verification, directly inserting such a subtraction when transforming the program is infeasible. Given the offset is a result of pointer arithmetic commands, *e.g.*, $w := x + y$, we can ensure the above predicate holds by enforcing that the offset of w is within the verifier's range after every arithmetic, *i.e.*, enforcing $e_l^{\#\prime}(w) <= e_n(w) <= e_h^{\#\prime}(w)$, where $e_l^{\#\prime}(w)$ and $e_h^{\#\prime}(w)$ are the verifier's range after the arithmetic. Therefore, we conduct the following enforcement:

$$\frac{e_t^{\#}(x) \in \mathcal{T}_{mem} \quad \neg(e_l^{\#\prime}(w) - e_l^{\#}(x) \leq e_n(y) \leq e_h^{\#\prime}(w) - e_h^{\#}(x))}{\langle w := x + y, \sigma \rangle \Rightarrow_{aee} \bot} \tag{26}$$

For $w := x + y$, when the verifier's tracked tag of x is a memory tag ($e_t^{\#}(x) \in \mathcal{T}_{mem}$), the semantics is adapted in P_{aee} to terminate the program when the value of y (the added offset) is not within following enforced range:

$$e_l^{\#\prime}(w) - e_l^{\#}(x) \leq e_n(y) \leq e_h^{\#\prime}(w) - e_h^{\#}(x)$$

Given the offset of the source operand x is already enforced:

$$e_l^{\#}(x) <= e_n(x) <= e_h^{\#}(x)$$

Hence, $e_l^{\#\prime}(w) <= e_n(w) = e_n(x) + e_n(y) <= e_h^{\#\prime}(w)$ holds, and the offset of w is enforced. Importantly, Equation 26 can be encoded directly with y (a scalar value) and the verifier's ranges (known during the transformation) without requiring the base address of w. In comparison with Equation 7, Equation 26 enforces that the added offset y must change the pointer from one tracked range to another tracked range. Similar enforcement also applies to the subtraction (Equation 8), both jointly enforce the pointer offsets.

Tag Enforcement. Next, we ensure the predicate $e_t(p) = e_t^{\#}(p) \in \mathcal{T}_{mem}$ holds, *i.e.*, enforcing that p stores an address with the same tag as tracked by the verifier. The tag of a value is determined after the assignment or the mem commands. Most commands, *e.g.*, the arithmetic and conditional commands, do not change the value tag according to the semantics shown in Figure 8, and thus the tracked tag by the verifier remains the same as that during the execution, and we do not need to insert assertions for those cases. However, the tag and offset may be altered after storing and loading an address value to and from a memory cell, since memory

accesses may overlap. The approximation may differ with runtime due to the verifier's soundness bugs in those cases. Therefore, for $w :=_{sz} *p$, we need to ensure $e(w)$ aligns with $e^{\#}(w)$, and the tag of `w` after loading must be the same as the verifier's tracked tag. The enforcement is represented by the following equations:

$$\frac{e_t^{\#}(x) \in \mathcal{T}_{mem} \quad (\mu', \delta') = Store'(\mu, \delta, (e(p), sz), e(x))}{\langle *p :=_{sz} x, \sigma \rangle \Rightarrow_{aee} (e, \mu', \delta'[(e(p), sz) \mapsto e^{\#}(x)])} \tag{27}$$

$$\frac{e_t^{\#}(w) \in \mathcal{T}_{mem} \quad \delta(e(p), sz) \neq \mu^{\#}(e(p), sz)}{\langle w :=_{sz} *p, \sigma \rangle \Rightarrow_{aee} \bot} \tag{28}$$

We extend the machine state with an additional map δ to associate memory cells with the verifier's tracked address value. Intuitively, δ mirrors the verifier's tracked value of memory cells during the execution, and the recorded value in δ is used to check against the tracked value when loading from cells. More concretely, if the verifier concludes the tag of a loaded value is a memory tag, then there must be a previous store command saving it to the cell. Therefore, when storing an address value to a memory cell, the verifier's tracked value is associated with the cell in δ (Equation 27) first. When performing normal memory store operations, the original $Store()$ helper is adjusted to take δ as input, and in addition to updating the memory state μ, it resets the overlapped cells in δ to reflect the effect of the store to the recorded values:

$$\begin{aligned} &Store'(\mu, \delta, c, (t, n)) = (\mu', \delta') \\ &\quad \text{where } \delta' = \delta[c_0 \mapsto \bot \mid c_0 \in dom(\delta) \wedge \overline{c_0} \cap \overline{c} \neq \emptyset] \\ &\quad\quad \mu' = Store(\mu, c, (t, n)) \end{aligned} \tag{29}$$

Subsequently, when loading to a register `w` and if the verifier believes an address is loaded into `w`, the verifier's tracked value of `w` is checked against the recorded value of the cell in δ (Equation 28), which enforces the two must be the same; otherwise, the program is aborted. Therefore, Equation 28 and Equation 27 enforce $e_t(p) = e_t^{\#}(p) \in \mathcal{T}_{mem}$ during execution.

Size Enforcement. Based on the aforementioned enforcement, the presumptions of Equation 24 are enforced; we need to further enforce $e_l^{\#}(y) \leq e_n(y) \leq e_h^{\#}(y)$ for Equation 25, *i.e.*, ensuring the value of the size register passed to a function call is bounded within the tracked range:

$$\frac{e_t^{\#}(x) \in \mathcal{T}_{mem} \quad \neg(e_l^{\#}(y) \leq e_n(y) \leq e_h^{\#}(y))}{\langle w := \texttt{func}(x, y), \sigma \rangle \Rightarrow_{aee} \bot} \tag{30}$$

For $w :=$ `func`(x, y), when the function accesses the memory pointed by `x`, Equation 30 enforces the size value in `y` to be within the verifier's tracked range.

The semantics of `store` and `func` in P_{aee} are adapted to update δ accordingly, and for all the other commands, the semantics under AEE remain the same as those shown in Figure 8, where δ keeps the same before and after the transition:

$$\frac{\langle cmd, (e, \mu) \rangle \Rightarrow (e', \mu')}{\langle cmd, (e, \mu, \delta) \rangle \Rightarrow_{aee} (e', \mu', \delta)} \tag{31}$$

Theorem 1. *Equation 22 and Equation 23 hold under the approximation enforcement, provided $InBounds^{\#}()$ is sound.*

Proof. Intuitively, Equation 27 and Equation 28 ensure that the tag of an address during the execution matches the verifier's tracked tag, Equation 26 ensures an offset within the tracked range. Therefore, each memory access is enforced to target a pointer with a restricted offset. Equation 30 enforces the size value passed to a call is within range. Hence, AEE is complete *w.r.t.* ensuring memory safety and the permitted execution is safe, assuming the safety check $InBounds^{\#}()$ is sound. More concretely, we prove each predicate in Equation 22 and Equation 23 holds, thereby proving the equations:

- For $e_t(p) \in \mathcal{T}_{mem}$, *only* storing and loading an address value to and from a memory cell may alter the tag, and Equation 28 and Equation 27 enforce $\delta(e(p), sz) = \mu_t^{\#}(e(p), sz) \wedge \mu_t^{\#}(e(p), sz) \in \mathcal{T}_{mem}$, *i.e.*, the tag during execution must match as the verifier tracked tag while the latter is a memory tag; hence, the predicate holds.
- For $0 \leq e_n(p) \leq sizeof(e_t(p)) - sz$, Equation 26 enforces all the offset added to or subtracted from the address value remain within the verifier's tracked range, thus $e_l^{\#}(p) \leq e_n(p) \leq e_h^{\#}(p)$ holds, *i.e.*, the pointer offset being within the verifier's range. Based on Equation 24, $InBounds^{\#}(e^{\#}(p), sz) \implies InBounds(e(p), sz)$ and the former holds, the predicate holds.
- For $0 \leq e_n(x) \leq sizeof(e_t(x)) - e_n(y)$, Equation 30 enforces the value of the size register remains within the tracked range for the call command. Similarly, $InBounds^{\#}(e^{\#}(x), e_n^{\#}(y)) \implies InBounds(e(x), e_n(y))$ and the former holds, hence the predicate holds.

□

5.3 Implementation

We implement our AEE prototype, which is publicly available [84], by modifying the eBPF verifier in the Linux kernel. Figure 11 illustrates three examples, one for each type of enforcement. In general, to apply AEE, one can first record the verifier's approximations and then transform the target program by adding enforcement assertions.

Our prototype involves two major parts: (1) recording the verifier's approximations during the verification in `struct bpf_insn_aux_data`, an auxiliary structure storing per-instruction state, and (2) inserting assertions in the program based on the recorded information, implemented as rewriting passes in `bpf_misc_fixups()`, a standard post procedure after the verification. The offset (Equation 26) and size enforcement (Equation 30) are realized as follows. Since offset values and pointer types are restricted to remain the same across paths for eBPF programs [34], ranges and tags are consistent. We record the verifier's tracked ranges for the pointer

offset and size register in the auxiliary structure and then insert eBPF instructions that check the runtime value against the collected range and enforce them before the corresponding pointer arithmetic and call instructions.

We utilize the ARM pointer authentication mechanism [33] to implement the conceptual map δ. The mechanism provides two primitives `pac()` and `aut()`, where the former signs a pointer with an additional modifier, and the latter decodes a signed value with the same modifier, and any changes in the signed value cause an authentication failure. In our approach, when the verifier concludes a pointer is spilled to memory, we embed the verifier's approximation into the pointer's signature, thus incurring no additional space overhead. Upon loading a pointer, the stored signature is authenticated to verify consistency with the verifier's approximation. We discuss the associated security implications in Section 7. More concretely, for a store command, if the verifier's tracked tag of the source register is a memory tag, we record the approximation in the auxiliary structure. During just-in-time (JIT) compilation, we then emit a hardware `pac()` instruction that signs the pointer (*i.e.*, Equation 27) using the tracked value. Next, for a load instruction, if the verifier's tracked tag for the corresponding stack slot is a memory tag, we mark the destination register for enforcement. In the JIT phase, we inject `aut()` to authenticate the pointer; any mismatch leads to an authentication failure, thereby implementing the tag enforcement. Finally, because the verifier limits tag tracking to stack operations, this rewriting pass only considers pointer spill and fill within the stack memory.

6 Empirical Results

In this section, we evaluate the effectiveness of AEE. All experiments were conducted on an Apple M1 Pro processor with 8 cores and 16 GiB of memory. The M1 processor was selected for its support of ARM pointer authentication. Our AEE implementation is based on the Linux kernel version 6.7. For evaluation, we compiled the kernel with our modifications using the standard eBPF configurations [20, 21].

6.1 Effectiveness of AEE

We evaluate the effectiveness of AEE by quantifying its trusted code base and applying it to defend against real malicious programs exploiting soundness bugs in the verifier.

Reduced Trust Base. The existing approach assumes the soundness of the verifier. In contrast, AEE minimizes this assumption by relying primarily on the verifier's safety-checking procedures. We quantify the lines of code in AEE's trust base and compare this with the existing approach.

The verifier ensures memory safety by tracking pointer offsets, size values, and tags across abstract domains, subsequently validating memory accesses based on this information. Consequently, the trust base of the existing approach encompasses both the approximation tracking and the associated checking procedures. AEE guarantees memory safety by enforcing pointer offsets, tags, and size values within the approximation, which does not depend on the tracking procedures but relies exclusively on the checking routines and AEE itself. The related approximations primarily involve range analysis, branch analysis, and stack state tracking. For the safety checks, the verifier mainly uses `check_mem_access()` and associated routines.

Program	Approximations	Runtime States
0: *(u64*)(r10 -40) = -1	0: fp-40 = u32_max	0: fp-40 = u64_max
1: r1 = *(u64*)(r10 -40)	1: r1 = u32_max	1: r1 = u64_max
2: r1 >>= 63	2: r1 = 0	2: r1 = 1
3: r1 += 1	3: r1 = 1	3: r1 = 2
4: r1 *= -8	4: r1 = -8	4: r1 = -16
5: r2 = r10	5: r2 = fp	5: r2 = fp
assert r1 == -8	…	Terminated
6: r2 += r1		

(a) Offset Enforcement. The verifier loses sign information after instruction #0, and at instruction #4, the tracked value of `r1` is -8, whereas the runtime value is -16. Since the access is only validated with `r1=-8`, the assertion inserted by AEE terminates the program upon the inconsistency, thereby preventing the exploit's attempt.

Program	Approximations	Runtime States
0: *(u64*)(r10 -40) = -1	0: fp-40 = u32_max	0: fp-40 = u64_max
1: r1 = *(u64*)(r10 -40)	1: r1 = u32_max	1: r1 = u64_max
2: r1 >>= 63	2: r1 = 0	2: r1 = 1
3: r1 += 1	3: r1 = 1	3: r1 = 2
4: r2 = r1	4: r2 = 1	4: r2 = 2
5: r1 = r10 -16	5: r1 = fp-16	5: r1 = fp-16
assert r2 == 1	…	Terminated
6: load(uptr, r1, r2)		

(b) Size Enforcement. The program invokes a helper at instruction #6, which loads external payloads into the region pointed to by `r1` with a size specified by `r2`. AEE enforces that the value of `r2` must be 1, the tracked value verified to be safe at instruction #4. The exploit attempts to corrupt a pointer with the helper call, but AEE aborts the execution when the size value diverges from the approximation.

Program	Approximations	Runtime States
0: r1 = map_ptr	0: r1 = map_ptr	0: r1 = map_ptr
pac(r1, map_ptr)	…	…
1: *(u64*)(r10 -9) = r1	1: fp-16 = map_ptr	1: fp-16 = T_s
2: r2 = *(u64*)(r10 -16)	2: r2 = map_ptr	2: r2 = T_s
aut(r2, map_ptr)	…	Terminated

(c) Tag Enforcement. Both the tracked and runtime tags of `r1` are identical at instruction #1. AEE inserts a `pac()` instruction before it to record the tag stored in the memory cell `fp-9`. Due to the soundness bug, the verifier incorrectly allows the pointer spill and mistakenly treats the tracked tag of `fp-16` as a memory tag, permitting subsequent unsafe accesses. The inconsistency is detected by `aut()`, preventing the unsafe dereference of `r2`.

Figure 11: Examples of AEE's enforcement: each subfigure shows an unsafe program, the unsound approximations, and runtime states. AEE terminates all the malicious programs, keeping the kernel unaffected.

Table 1: Routines related to ensuring memory safety. The existing approach assumes the soundness of both the checking and tracking routines, totaling 5,068 lines of code, while AEE relies mainly on the safety check, achieving a 4.5x reduction.

Routine/Component	Description	Lines of Code	In Trust Base
AEE Routines	Enforce pointer offset, tag, and size approximations	258	✓
`check_mem_access()`	General routine invoking specific access checks based on region type	240	✓
`check_mem_region_access()`	Validates memory region access using variable offsets for specified sizes	53	✓
`check_stack_access_within_bounds()`	Ensures stack accesses are within allocated bounds	57	✓
Other Access Checks	Additional checks for various region types, including `check_map_access()`, *etc.*	517	✓
Range Analysis	Determines ranges of pointer offsets and scalar values across five abstract domains	1,824	✗
Branch Analysis	Identifies execution paths and updates ranges of scalars or offsets	1,372	✗
Stack State Tracking	Tracks the contents of each stack slot following load and store instructions	1,005	✗

Table 1 presents the relevant routines. The existing approach assumes the soundness of these routines, totaling 5,068 lines of code. In contrast, AEE relies only on its rewrite passes and the safety checks, achieving a 4.5x reduction in the trust base. Generally, tracking program states across different abstract domains is complex, amounting to 4,201 lines of code, whereas performing safety checks based on the approximations is relatively straightforward, involving merely 867 lines, as shown in Table 1. By enforcing approximations, AEE significantly reduces the trust base.

Improved Safety. We evaluate AEE's capability to maintain kernel integrity even in the presence of soundness bugs outside the trust base. To this end, we collected all *publicly available* malicious eBPF programs that exploit unsound approximations over the past five years, resulting in a set of seven programs, each aiming to achieve local privilege escalation (LPE). Table 2 lists these soundness bugs. We executed these exploit programs to observe whether AEE terminates their execution, thereby eliminating potential security issues.

With the existing approach, these malicious programs were loaded and executed due to the verifier's soundness bugs, ultimately achieving LPE. These vulnerabilities arise because soundness bugs in the verifier's approximations allow concrete states to extend beyond intended bounds. Malicious programs exploit this discrepancy to generate non-contained states and craft unsafe operations on them, while the verifier erroneously validates these operations using unsound approximations. In contrast, AEE permits the loading of these programs but terminates their execution via the inserted assertions once the runtime state diverges from the approximation. Consequently, AEE effectively prevents all malicious attempts by them. Figure 11 illustrates three examples.

Future Exploits. Although our evaluation is constrained by the number of publicly available malicious programs, AEE can defend against future malicious programs exploiting bugs outside the trust base (Table 1). We have provided a formalization of AEE by specifying its effects within the operational semantics. Supported by our proofs, the formalization demonstrates that our application of AEE is complete by design *w.r.t.* ensuring memory safety under the reduced trust base, *i.e.*, further malicious programs will be terminated, once escaped execution with non-contained states occur.

Table 2: Soundness bugs in the state tracking with publicly available exploits over the past five years.

CVE/Fix Commit	Bug Description
CVE-2020-8835	Incorrect optimization of arithmetic operations
CVE-2020-27194	Use of incorrect range in an abstract domain
CVE-2021-3490	Improper extension of upper 32-bits
CVE-2021-31440	Faulty range refinement procedures
CVE-2022-23222	Incorrect nullable pointer arithmetic
CVE-2023-2163	Improper pruning of unsafe states
811c363645b3	Incorrect immediate value spilling

6.2 Performance Impact

We evaluate AEE's overhead by measuring its impact on verification time, binary size, and execution time. To evaluate each enforcement, we apply them individually and load the programs with and without AEE for comparison.

Dataset. To comprehensively evaluate AEE, we compiled a diverse dataset of 1,141 eBPF programs, averaging 1,139 bytes in program size, including (1) `Linux-Progs`: general-purpose eBPF programs collected from the Linux repository [23] and the dataset by Gershuni *et al.* [51], (2) `Linux-Bench` [19]: benchmark programs to assess memory operation throughput, (3) `Filter` [27]: CPU-intensive packet filtering programs with standard workload, and (4) `Katran` [31]: a production, performance-sensitive load balancer. This dataset encompasses representative real-world programs covering various scenarios, ensuring a robust evaluation.

Binary Size. Table 3 presents AEE's impact on binary size. The *All* column indicates that all enforcement mechanisms are enabled, while the remaining columns correspond to individual enforcement. With all enforcements enabled, 28.8% (329 programs) exhibit an increase in binary size, while the rest of the programs remain unaffected. Although the highest observed increase is 30.3%, the absolute size increment

Table 3: AEE's impact on binary size. *Impacted* shows the percentage and count of programs affected by the enforcement. *Max* represents the maximum increase in absolute and percentage terms. *Impacted avg* illustrates the average increase for the impacted programs, while *Overall avg* shows the average increase considering all programs.

	All	Offset	Tag	Size
Impacted (%)	28.8% (329)	6.8% (77)	6.4% (73)	22.5% (257)
Max (%)	30.3% (80/264)	22.2% (32/144)	30.3% (80/264)	22.2% (32/144)
Impacted Avg	4.8%	1.6%	0.7%	2.6%
Overall Avg	1.4%	0.5%	0.2%	0.7%

remains minimal at 80 bytes. The average increase among affected programs is 4.8%, while the overall average increase, considering both affected and unaffected programs, is 1.4%.

The low impact on binary size introduced by AEE aligns with our expectations, which can be attributed to three primary factors. First, arithmetic instructions in eBPF programs predominantly operate between scalar values or between a pointer and a constant offset, *e.g.*, sourced from struct field access. Offset enforcement only inserts assertions before arithmetic instructions involving a pointer and a variable. Second, spilling and filling pointers to and from the stack are restricted to users with specific capabilities [45, 83], resulting in infrequent occurrences and thus requiring fewer tag enforcements. Third, the proportion of call instructions within programs is low, and only a limited number of helper functions incorporate size parameters [22]. Collectively, these factors contribute to the negligible increase in binary size observed with AEE.

Execution Time. Measuring the runtime overhead of eBPF programs poses several challenges: (1) they are typically small and execute within nanoseconds in the kernel context, and (2) they are attached to various hooks and can only be triggered by specific workloads. To address these challenges and minimize potential inaccuracies, we leverage existing infrastructure, workloads, and benchmarks to evaluate AEE.

Workload Setup. For programs in `Linux-Progs` with available driving workloads, we execute them in tight loops and utilize eBPF's performance monitoring support [25] to measure average execution time. This approach was applied to a subset of 212 programs within this dataset; the remaining programs were loaded only due to the absence of associated workloads. For programs in `Linux-Bench`, `Katran`, and `Filters`, we employ the provided benchmark suites to assess the overhead. In `Linux-Bench`, the benchmark program loads the extensions, frequently triggers their execution, and measures memory access throughput. For `Filters`, following prior work [60, 61, 95], we load the filter programs and set up a server and client that continuously transfer packets to trigger these filters, subsequently measuring packet throughput. Finally, we evaluate the throughput of the eBPF-based load balancer using the driving workload provided by Katran's `-perf_testing` option in its benchmark suite [32]. This ensures AEE's overhead is evaluated across varied workloads.

Table 4: AEE's runtime overhead, measured in nanoseconds (*ns*) for `Linux-Progs` and `Katran` (*per packet*), thousand operations per second (*tops*) for `Linux-Bench`, and thousand packets per second (*tpps*) for `Filters`. A dash (–) indicates that no programs in the benchmark require the enforcement.

Benchmark	All	Offset	Tag	Size
Linux-Progs (*ns*)	174.1 (1.2%)	173.6 (0.2%)	173.6 (0.2%)	173.7 (0.5%)
Linux-Bench (*tops*)	1126 (1.1%)	1136 (0.3%)	1127 (0.9%)	1133 (0.5%)
Katran (*ns/pkt*)	130.2 (1.1%)	129.1 (0.2%)	130.0 (1.0%)	–
Filters (*tpps*)	99.0 (1.2%)	–	99.5 (0.7%)	100.0 (0.3%)

Table 5: Impact on verification time. *Max* shows both the percentage and the absolute time increase. While the maximum percentage increase is high, the absolute time remains low.

	All	Offset	Tag	Size
Min	0.5%	0.5%	0.1%	0.1%
Max (µs)	85.0% (106/125)	80.7% (253/314)	30.2% (72/240)	84.0% (182/217)
Impacted Avg	21.3%	6.5%	0.8%	15.6%
Overall Avg	4.8%	1.5%	0.2%	3.6%

Table 4 illustrates the impact of AEE on program execution time. Overall, AEE incurs an average increase of 1.2% across all benchmarks when all enforcement types are enabled, with a maximum observed increase of 4.5%. For instance, in the `Linux-Progs` benchmark, the overall average increase is 1.2%, while individual enforcement types contribute even less, with the offset and tag enforcement adding 0.2% each. These results are consistent with the increase in binary size and can be attributed to two primary factors: (1) eBPF programs are restricted to simple bounded loops [1], low branch complexity [5], and a constrained total instruction count [4], ensuring that the inserted instructions reside in trivial paths that terminate quickly; and (2) the additional CPU cycles required for the inserted instructions are minimal, with the worst-case `pac()`/`aut()` operations necessitating only 5–7 additional cycles [65, 96]. The consistently low overhead underscores AEE's efficiency in enhancing safety guarantees without compromising performance.

Verification Time. Table 5 presents AEE's impact on the verification time. The average increase for all the impacted programs is 21.3%, while the overall average increase across all programs is 4.8%. Although the maximum observed increase reaches 85.0%, this does not significantly impact AEE's scalability for two major reasons: (1) the absolute increase remains minimal, at the microsecond level, and (2) program loading is infrequent, with verification being a one-time operation. Therefore, AEE maintains efficient scalability.

7 Discussion

7.1 Pointer Authentication

We implement the tag enforcement using signature-based techniques to inline tags, and pointer authentication seamlessly integrates into our prototype. Key security considerations are

as follows. First, an attacker might attempt to brute-force the pointer authentication code (PAC) or the associated key. However, the likelihood of guessing the correct code is low due to the large space [65], and each failed attempt triggers an observable abnormal termination. Second, if an attacker gains control over pointers or the modifier used for signing, they could theoretically generate a malicious PAC. Under our threat model, however, these events are precluded because only kernel pointers are signed, and the modifier is securely provided by the verifier. Third, reusing a signed pointer may appear; yet, since the reused pointer carries the same tag, subsequent accesses remain safe.

On other platforms, similarly efficient methods can be used. For instance, one could maintain a secret salt in the kernel and compute a hash from the pointer and the verifier's tracked value. This hash could then be stored in the higher-order bits of the pointer, leveraging the upper-bit ignore feature common to many architectures. In addition, because pointer spilling and filling occur infrequently (0.7%, as shown in Table 3), the performance impact remains consistent.

7.2 Generality

We illustrate AEE's generality through additional vulnerabilities and discuss its potential for other guarantees in eBPF.

CVE-2021-4001. This vulnerability arises from a data race between the verifier and the map runtime utility, resulting in a Time-of-Check to Time-of-Use (TOCTOU) condition [13]. eBPF programs can access maps shared with user space, and when these maps are mutable, the verifier makes no assumptions about their contents, thus marking the target register as an unbounded integer. However, when a map is made read-only through the irreversible `map_freeze()` (and other relevant flags [7]), as depicted in Figure 12, the verifier retrieves the value stored in it and marks the register as this constant, facilitating subsequent analysis. The data race permits concurrent modification of ostensibly frozen maps, invalidating this assumption (`r1` can take on arbitrary values at runtime) and enabling attackers to compromise kernel memory.

TOCTOU exploits fundamentally rely on state inconsistency, a vulnerability that AEE can mitigate by enforcing strict adherence to verifier approximations. In this case, the verifier concludes that the register holds a constant. When attackers exploit the race condition to alter the value, AEE detects the deviation and aborts execution. As illustrated in the lower section of Figure 12: (1) If the value is used as a pointer offset in direct memory access, offset enforcement captures inconsistencies during pointer arithmetic; (2) If it is used as a size parameter in helper calls for indirect access, size enforcement intervenes accordingly; and (3) alternatively, AEE can be preemptively adapted to enforce immutability by substituting map loads with constant moves during transformation.

In practice, eBPF ensures memory safety, termination, and no information leakage. For termination, the verifier provides

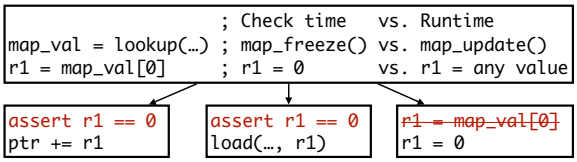

Figure 12: CVE-2021-4001 and AEE's enforcement for it.

```
r0 = lookup(…)          ; r0 = ptr(size=16)
r1 = *(u64*)(fp-off)    ; unsound approximation of r1
                        ; r1 = 0 vs. r1 = 4
assert r1 == 0          ; abort
r2 = *(u64*)(r0 + r1)   ; safe access with r1=4
```

```
r0 = lookup(…)          ; r0 = ptr(size=16)
r1 = *(u64*)(fp-off)    ; r1 = 32 vs. r1 = 32
assert r1 == 32         ; permit
r2 = *(u64*)(r0 + r1)   ; unsound safety check
                        ; unsafe access not rejected
```

Figure 13: Illustrative examples for AEE's execution effects.

safe loop bounds, and AEE can enforce them via dynamic loop counters. For information leakage, AEE enforces value tags, and the verifier ensures addresses are not written to user-accessible regions. Broadly, one may adapt the operational semantics, identify relevant operations and approximations, and rewrite extensions to enforce additional properties.

7.3 Execution Effects

AEE may terminate safe programs due to unsound approximations (#2 in Figure 10). The upper section of Figure 13 illustrates this case: (1) the program accesses a region safely using a valid offset, (2) the offset is incorrectly tracked, (3) the safe program is accepted by the unsound approximation, and AEE enforces it. At runtime, AEE detects this discrepancy and halts the execution. While seemingly erroneous, such termination effectively exposes latent verifier flaws that could lead to vulnerabilities. Without AEE, such defects remain unnoticed. In practice, extensions are tested against offline workloads before deployment. Early AEE terminations uncover verifier weaknesses promptly before impacting production systems.

Conversely, AEE might permit unsafe programs when the approximation ($V.a$) is sound, but the associated safety check ($V.s$) is flawed (#3 in Figure 10). The lower section of Figure 13 shows a scenario where the verifier correctly tracks the range of `r1`, and the runtime value aligns with the statically checked one: the execution with `r1=32` is permitted. However, if an unsound safety check erroneously deems the final out-of-bounds access as safe, this violation goes undetected: AEE enforces $V.a$ and presumes $V.s$ to be correct. Nonetheless, such cases are rare. The approximation logic $V.a$ involves complex abstract interpretation over instruction semantics and diverse abstract domains, making it more susceptible to errors. In contrast, safety checks $V.s$ are much simpler, involving direct validations over computed approximations, which renders them easier to implement and audit.

Table 6: Comparison of AEE with the existing work. Compatibility legend: ✓ = no infrastructure changes, △ = requires runtime support only, ✗ = requires substantial modifications.

	Mechanism	Granularity	Kernel object access	Compatibility
SandBPF [66]	Software Isolation	Page	Mirror copy	✗
MOAT [67]	Intel MPK	Page	Dynamic remap	✗
HIVE [99]	AArch64 LSU	Page	Dynamic patching	✗
BeeBox [61]	Sandboxed space	Page	Selective copy	✗
KFlex [49]	Address masking	Page/Object	Native	△
AEE	Approximation enforcement	Object	Native	✓

8 Related Work

eBPF Hardening. Previous work has explored mechanisms to enhance the security of eBPF extensions [54]. SandBPF [66] confines memory accesses to specific pages allocated for extensions. MOAT [67] utilizes Intel Memory Protection Keys (MPK) [29], while HIVE [99] uses AArch64 unprivileged load/store (LSU) instructions to restrict accesses to sanctioned memory areas. BeeBox [61] tackles transient execution attacks by rerouting memory accesses into an isolated sandbox. KFlex [49] adopts address masking for extension-owned memory while relying on the verifier for other properties.

Whereas existing work mainly isolates extensions, AEE maintains extensions and the kernel in the same domain by providing a hybrid safety guarantee. AEE relies on the safety checks and its enforcement logic (Section 7.3), while existing work mainly depends on the isolation logic. Technically, AEE exhibits various benefits, as detailed in Table 6. Unlike the page-level protections, AEE operates at the object level. AEE maintains compatibility, whereas MOAT and HIVE require significant alterations, including restructuring the map layout. Isolation requires dynamically copying kernel objects into extension-accessible regions, which AEE completely avoids.

General SFI. Software fault isolation techniques are adopted to achieve various guarantees [44, 55, 68, 70, 73, 80, 87, 93], such as AddressSanitizer (ASan) [81] and its variant KASan. Generally, SFI-based systems share a common structure: (1) they define an isolated domain (*e.g.*, ASan's shadow memory), (2) maintain metadata (*e.g.*, `asan_poison()`), and (3) perform runtime checks (*e.g.*, `asan_load/store()`) against the metadata. AEE decouples these steps across stages: (1) the verifier *statically* gathers approximations; (2) enforcement ensures adherence to them; and (3) safety is assured through trusted, static checks rather than extensive runtime checks.

AEE delivers fine-grained protection. ASan's shadow memory lacks the originally associated region of each pointer: if an out-of-bounds access strays into a different region, ASan fails to detect the violation. AEE enforces approximations at the object level and reliably detects such cross-region violations. Second, when an extension invokes helper functions that cross the extension-kernel boundaries, full kernel instrumentation is required by KASan. AEE avoids this overhead by confining enforcement solely within the extension, leaving the kernel unmodified. Finally, leveraging the precomputed approximations and trusted static checks, AEE requires fewer runtime checks as illustrated in Figure 14.

```
r0 = lookup(…)
asan_poison(r0, …)
r0 += off
asan_load(r0, …)
r1 = *(u64*)(r0 + 8)
asan_store(r0, …)
*(u64*)r0 = 0
```

```
…
assert off == checked_off
r0 += off
; statically checked by V.s
r1 = *(u64*)(r0 + 8)
; statically checked by V.s
*(u64*)r0 = 0
```

Figure 14: ASan requires three runtime checks for this extension, while AEE needs one enforcement, and two trusted checks that *statically* ensure the safety of the checked offset.

Safe Languages. Several studies have explored designing safe kernel-extension languages [39, 79]. Jia *et al.* [58], for example, advocate writing kernel extensions in Rust, using a trusted userspace toolchain. SafeDrive [101] incorporates predefined annotations that enable the compiler to generate inline runtime checks, and SPIN [40] enforces explicit interface boundaries through compiler-level mechanisms. These approaches introduce an additional trust component in the compiler. In comparison, AEE ensures the compatibility while maintaining a reduced trust base.

Extension Verification. Necula introduced proof-carrying code (PCC) [74], wherein extensions are accompanied by proofs, which are validated by a proof checker before execution. Nelson *et al.* [76] extended PCC to eBPF, while XFI [50] implemented a similar approach for Windows binaries. However, generating safety proofs is inherently challenging, and existing prototypes often struggle to provide comprehensive guarantees with acceptable proof sizes. eBPF adopts the verifier as a gatekeeper; following this approach, Gershuni *et al.* [51] proposed a verifier based on the abstract interpretation framework. AEE builds upon the verifier but combines its safety checks with approximation enforcement.

Verifier Soundness. Substantial efforts have been devoted to improving the verifier's soundness through formal verification [75, 90–92, 94, 97] and dynamic testing [56, 72, 78, 82, 85, 86, 88, 98]. Formal verification ensures the consistency between the implementation and its specification, while testing relies on well-crafted oracles to uncover errors. AEE complements the existing solutions by providing an effective mitigation strategy for unsound approximations, ensuring safe execution even in the presence of verifier flaws.

9 Conclusion

In this paper, we have proposed Approximation-Enforced Execution, a novel concept to achieve the safe execution of untrusted kernel extensions. Rather than using the verifier as a gatekeeper, AEE provides safety guarantees by only trusting its safety checks and enforcing strict adherence to the checked approximations. Our evaluation has demonstrated AEE's effectiveness by achieving improved safety guarantees with a drastically reduced trust base while incurring low overhead.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback on an earlier version of this paper. We are especially grateful to our shepherd for the prompt and constructive guidance that helped strengthen our paper and improve its exposition. This work was partially supported by a research award from the eBPF Foundation, which we appreciate and acknowledge.

10 Ethics Considerations

In this study, we have carefully evaluated the ethical implications of our work. Our primary contribution, AEE, enhances system integrity by ensuring the safe execution of untrusted kernel extensions and serving as a mitigation strategy against unsound verifier approximations. All soundness bugs discussed in this paper are already publicly disclosed and have been patched in affected kernel versions. Consequently, the exploit programs employed in our experiments target only these known bugs, posing no new threat to current systems.

All experimental activities reported in Section 6 were performed on private machines within a controlled laboratory setting. We executed exploit programs in isolated environments, such as virtual machines, with full awareness of their behavior. This containment strategy prevented any unintended negative impact on production or external systems.

11 Open Science

We fully endorse the principles of open science and recognize the value of transparent and reproducible research. Accordingly, we have released all relevant artifacts related to our work [84], including code, data, and detailed documentation. We have also participated in artifact evaluation to facilitate community engagement and promote rigorous assessment of our research.

References

[1] Bounded Loop in eBPF. https://docs.ebpf.io/linux/concepts/loops.

[2] bpf: Fix incorrect delta propagation between linked registers. https://lore.kernel.org/bpf/20241016134913.32249-1-daniel@iogearbox.net/.

[3] bpf: Fix truncation bug in coerce_reg_to_size_sx(). https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf.git/commit/?id=ae67b9fb8c4e.

[4] BPF Instructions Limit. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/linux/bpf.h?h=v6.10#n1900.

[5] BPF Jump Complexity Limit. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/kernel/bpf/verifier.c?h=v6.7#n187.

[6] bpf: Track equal scalars history on per-instruction level. https://github.com/torvalds/linux/commit/2b7350d7ca65.

[7] Conditions Determining the Map Mutability in eBPF. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/kernel/bpf/verifier.c?h=v6.15-rc7&id=a5806cd506af5a7c19bcd596e4708b5c464bfd21#n6928.

[8] CVE-2020-27194. https://nvd.nist.gov/vuln/detail/CVE-2020-27194.

[9] CVE-2020-8835. https://nvd.nist.gov/vuln/detail/CVE-2020-8835.

[10] CVE-2021-31440. https://nvd.nist.gov/vuln/detail/CVE-2021-31440.

[11] CVE-2021-33200. https://nvd.nist.gov/vuln/detail/CVE-2021-33200.

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

[13] CVE-2021-4001. https://nvd.nist.gov/vuln/detail/CVE-2021-4001.

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

[15] CVE-2023-2163. https://nvd.nist.gov/vuln/detail/CVE-2023-2163.

[16] CVE-2024-41003. https://nvd.nist.gov/vuln/detail/CVE-2024-41003.

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

[18] eBPF. https://ebpf.io.

[19] eBPF Benchs. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/tools/testing/selftests/bpf/benchs?h=v6.7.

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

[21] eBPF Build Config on aarch64. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/tools/testing/selftests/bpf/config.aarch64?id=0dd3ee311255.

[22] eBPF Helper Calls. https://man7.org/linux/man-pages/man7/bpf-helpers.7.html.

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

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

[25] Enable eBPF Runtime Statistics Gathering. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/uapi/linux/bpf.h?h=v6.7#n792.

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

[27] Filter Programs. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/samples/bpf?h=v6.7.

[28] Incorrect immediate value stack spilling. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=811c363645b3.

[29] Intel 64 and IA-32 Architectures Software Developer Manuals.

[30] IO Visor Project. https://www.iovisor.org/technology/bcc.

[31] Katran Load Balancer. https://github.com/facebookincubator/katran/blob/def2e01b3aab8d4a1f27df8bb49267acd0980eca/katran/lib/bpf/balancer.bpf.c.

[32] Katran Tester. https://github.com/facebookincubator/katran/blob/def2e01b3aab8d4a1f27df8bb49267acd0980eca/katran/lib/testing/katran_tester.cpp.

[33] Pointer Authentication on ARMv8.3. https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf.

[34] Pointer Type and Offet Range Restrictions Across Paths in eBPF. https://elixir.bootlin.com/linux/v6.7/source/kernel/bpf/verifier.c#L12728.

[35] bpf: Fix incorrect sign extension in check_alu_op(). https://github.com/torvalds/linux/commit/95a762e2c8c942780948091f8f2a4f32fce1ac6f, 2017.

[36] bpf, x32: Fix bug with ALU64 LSH, RSH, ARSH BPF_X shift by 0. https://github.com/torvalds/linux/commit/68a8357ec15bdce55266e9fba8b8b3b8143fa7d2, 2019.

[37] bpf: Fix truncation bug in coerce_reg_to_size_sx(). https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=ae67b9fb8c4e, 2024.

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

[39] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamari, and Leonid Ryzhyk. System programming in rust: Beyond safety. *SIGOPS Oper. Syst. Rev.*, 51(1):94–99, sep 2017.

[40] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, page 267–283, New York, NY, USA, 1995. Association for Computing Machinery.

[41] 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.

[42] Ashish Bijlani and Umakishore Ramachandran. Extension framework for file systems in user space. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 121–134, Renton, WA, July 2019. USENIX Association.

[43] Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, and Peter Müller. Automatically Testing Implementations of Numerical Abstract Domains. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, page 768–778, 2018.

[44] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM*

SIGOPS 22nd Symposium on Operating Systems Principles, SOSP '09, page 45–58, New York, NY, USA, 2009. Association for Computing Machinery.

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

[46] 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*, POPL '77, page 238–252, 1977.

[47] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of generalized type unions. In *Proceedings of an ACM Conference on Language Design for Reliable Software*, page 77–94, New York, NY, USA, 1977. Association for Computing Machinery.

[48] Milo Craun, Adam Oswald, and Dan Williams. Enabling ebpf on embedded systems through decoupled verification. In *Proceedings of the 1st Workshop on EBPF and Kernel Extensions*, eBPF '23, page 63–69, New York, NY, USA, 2023. Association for Computing Machinery.

[49] Kumar Kartikeya Dwivedi, Rishabh Iyer, and Sanidhya Kashyap. Fast, flexible, and practical kernel extensions. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 249–264, New York, NY, USA, 2024. Association for Computing Machinery.

[50] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. Xfi: software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 75–88, USA, 2006. USENIX Association.

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

[52] Yi He, Roland Guo, Yunlong Xing, Xijia Che, Kun Sun, Zhuotao Liu, Ke Xu, and Qi Li. Cross container attacks: The bewildered eBPF on clouds. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5971–5988, Anaheim, CA, August 2023. USENIX Association.

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

[54] Kaiming Huang, Jack Sampson, Mathias Payer, Gang Tan, Zhiyun Qian, and Trent Jaeger. SoK: Challenges and Paths Toward Memory Safety for eBPF. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 810–828. IEEE Computer Society, 2025.

[55] Yongzhe Huang, Vikram Narayanan, David Detweiler, Kaiming Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. KSplit: Automating device driver isolation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 613–631, Carlsbad, CA, July 2022. USENIX Association.

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

[57] Jinghao Jia, Michael V. Le, Salman Ahmed, Dan Williams, and Hani Jamjoom. Practical and flexible kernel cfi enforcement using ebpf. In *Proceedings of the 1st Workshop on EBPF and Kernel Extensions*, eBPF '23, page 84–85, New York, NY, USA, 2023. Association for Computing Machinery.

[58] Jinghao Jia, Raj Sahu, Adam Oswald, Dan Williams, Michael V. Le, and Tianyin Xu. Kernel extension verification is untenable. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HOTOS '23, page 150–157, New York, NY, USA, 2023. Association for Computing Machinery.

[59] Jinghao Jia, YiFei Zhu, Dan Williams, Andrea Arcangeli, Claudio Canella, Hubertus Franke, Tobin Feldman-Fitzthum, Dimitrios Skarlatos, Daniel Gruss, and Tianyin Xu. Programmable system call security with ebpf, 2023.

[60] Di Jin, Vaggelis Atlidakis, and Vasileios P. Kemerlis. EPF: Evil packet filter. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 735–751, Boston, MA, July 2023. USENIX Association.

[61] Di Jin, Alexander J. Gaidis, and Vasileios P. Kemerlis. BeeBox: Hardening BPF against transient execution attacks. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 613–630, Philadelphia, PA, August 2024. USENIX Association.

[62] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-defined scheduling across the stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 605–620, New York, NY, USA, 2021. Association for Computing Machinery.

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

[64] Christian Klinger, Maria Christakis, and Valentin Wüstholz. Differentially Testing Soundness and Precision of Program Analyzers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 239–250, 2019.

[65] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. Pac it up: towards pointer integrity using arm pointer authentication. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 177–194, USA, 2019. USENIX Association.

[66] Soo Yee Lim, Xueyuan Han, and Thomas Pasquier. Unleashing unprivileged ebpf potential with dynamic sandboxing. In *Proceedings of the 1st Workshop on EBPF and Kernel Extensions*, eBPF '23, page 42–48, New York, NY, USA, 2023. Association for Computing Machinery.

[67] Hongyi Lu, Shuai Wang, Yechang Wu, Wanning He, and Fengwei Zhang. MOAT: Towards Safe BPF Kernel Extension, 2023.

[68] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Software fault isolation with api integrity and multi-principal modules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 115–128, New York, NY, USA, 2011. Association for Computing Machinery.

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

[70] Derrick Paul McKee, Yianni Giannaris, Carolina Ortega, Howard E Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. Preventing kernel hacks with hakcs. In *NDSS*, pages 1–17, 2022.

[71] Antoine Miné. Abstract domains for bit-level machine integer and floating-point operations. In *WING'12-4th International Workshop on invariant Generation*, page 16, 2012.

[72] Mohamed Husain Noor Mohamed, Xiaoguang Wang, and Binoy Ravindran. Understanding the security of linux ebpf subsystem. In *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '23, page 87–92, New York, NY, USA, 2023. Association for Computing Machinery.

[73] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev. LXDs: Towards isolation of kernel subsystems. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 269–284, Renton, WA, July 2019. USENIX Association.

[74] George C. Necula and Peter Lee. Safe kernel extensions without Run-Time checking. In *USENIX 2nd Symposium on OS Design and Implementation (OSDI 96)*, Seattle, WA, October 1996. USENIX Association.

[75] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. Specification and Verification in the Field: Applying Formal Methods to BPF Just-in-Time Compilers in the Linux Kernel. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, 2020.

[76] Luke Nelson, Xi Wang, and Emina Torlak. A proof-carrying approach to building correct and flexible in-kernel verifiers. In *Linux Plumbers Conference*, 2021.

[77] Jan Onderka and Stefan Ratschan. Fast Three-Valued Abstract Bit-Vector Arithmetic. In *Verification, Model Checking, and Abstract Interpretation: 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16–18, 2022, Proceedings*, page 242–262, 2022.

[78] Chaoyuan Peng, Muhui Jiang, Lei Wu, and Yajin Zhou. Toss a fault to bpfchecker: Revealing implementation flaws for ebpf runtimes with differential fuzzing. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 3928–3942, New York, NY, USA, 2024. Association for Computing Machinery.

[79] Matthew J. Renzelmann and Michael M. Swift. Decaf: moving device drivers to a modern language. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, page 14, USA, 2009. USENIX Association.

[80] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: surviving misbehaved kernel extensions. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, page 213–227, New York, NY, USA, 1996. Association for Computing Machinery.

[81] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *2012 USENIX annual*

technical conference (USENIX ATC 12), pages 309–318, 2012.

[82] Sven Smolka, Jens-Rene Giesen, Pascal Winkler, Oussama Draissi, Lucas Davi, Ghassan Karame, and Klaus Pohl. Fuzz on the beach: Fuzzing solana smart contracts. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 1197–1211, New York, NY, USA, 2023. Association for Computing Machinery.

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

[84] Hao Sun. Approximation Enforced Execution of Untrusted Linux Kernel Extensions, June 2025. https://doi.org/10.5281/zenodo.15609051.

[85] Hao Sun and Zhendong Su. Validating the ebpf verifier via state embedding. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2024.

[86] Hao Sun, Yiru Xu, Jianzhong Liu, Yuheng Shen, Nan Guan, and Yu Jiang. Finding correctness bugs in ebpf verifier with structured and sanitized program. 2024.

[87] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. *SIGOPS Oper. Syst. Rev.*, 37(5):207–222, oct 2003.

[88] Jubi Taneja, Zhengyang Liu, and John Regehr. Testing Static Analyses for Precision and Soundness. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2020, page 81–93, 2020.

[89] 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.

[90] Harishankar Vishwanathan. Developing verified static analyzers for kernel extensions: A related work report.

[91] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. Sound, Precise, and Fast Abstract Interpretation with Tristate Numbers. In *CGO '22*, pages 254–265, 2022.

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

[93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. SOSP '93, page 203–216, New York, NY, USA, 1993. Association for Computing Machinery.

[94] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. Jitk: A trustworthy In-Kernel interpreter infrastructure. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 33–47, Broomfield, CO, October 2014. USENIX Association.

[95] Zhenyu Wu, Mengjun Xie, and Haining Wang. Swift: A fast dynamic packet filter. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08)*, San Francisco, CA, April 2008. USENIX Association.

[96] Sungbae Yoo, Jinbum Park, Seolheui Kim, Yeji Kim, and Taesoo Kim. In-Kernel Control-Flow integrity on commodity OSes using ARM pointer authentication. In *USENIX Security 22*, pages 89–106, Boston, MA, August 2022. USENIX Association.

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

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

[99] Peihua Zhang, Chenggang Wu, Xiangyu Meng, Yinqian Zhang, Mingfan Peng, Shiyang Zhang, Bing Hu, Mengyao Xie, Yuanming Lai, Yan Kang, and Zhe Wang. HIVE: A hardware-assisted isolated execution environment for eBPF on AArch64. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 163–180, Philadelphia, PA, August 2024. USENIX Association.

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

[101] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. Safedrive: safe and recoverable extensions using language-based techniques. In *Proceedings of OSDI '06*, page 45–60, USA, 2006. USENIX Association.