





# Comparing the Precision of Abstract Operators in the eBPF Verifier using Differential Synthesis

Matan Shachnai <sup>\*</sup>, Harishankar Vishwanathan ,  
Srinivas Narayana , and Santosh Nagarakatte 

Rutgers University, USA  
{m.shachnai, harishankar.vishwanathan,  
srinivas.narayana, santosh.nagarakatte}@rutgers.edu

**Abstract.** The eBPF verifier ensures the safety of user-supplied programs before they are executed in the Linux kernel, relying on abstract interpretation. While the verifier’s analysis must be sound, its utility hinges on precision. An overly conservative abstract operator can routinely cause the verifier to reject safe programs. In this paper, we introduce a framework for systematically comparing and validating the precision of competing abstract operator implementations used within the verifier. We provide a formal specification of the precision relationship between two abstract operators across all valid abstract inputs. However, reasoning about all valid abstract inputs over-approximates what is actually reachable in real verifier executions. This is because the eBPF verifier performs verification from a specific set of initial abstract states. Hence, many abstract inputs used in theoretical comparisons may never arise in practice. To address this gap, we propose SMT-based program synthesis to automatically generate concrete eBPF witness programs, explicitly demonstrating observable precision differences in actual verifier executions. Using these techniques and tools, we crafted a more precise multiplication abstract operator in the verifier, `bpf_mul`. Our multiplication patch has been upstreamed to the Linux kernel where the witness produced by our approach provided demonstration to the kernel developers. We have also used these techniques to check the precision of numerous kernel patches related to abstract operators in the eBPF verifier.

**Keywords:** Abstract interpretation · Kernel extensions · Program synthesis · eBPF

## 1 Introduction

The eBPF ecosystem has become the de facto approach for extending the functionality of the Linux kernel, enabling versatility, portability, and performance. It is used in a variety of contexts, such as load balancing [17], access control and DDoS mitigation [19, 37], tracing [74], and memory optimization [47]. A key

---

<sup>\*</sup> Corresponding author.

feature of the eBPF ecosystem is its safety guarantees; eBPF employs a static analyzer, the eBPF verifier, which serves as a bulwark to prevent incorrect eBPF programs from crashing or compromising the kernel once they are incorporated into the address space of the kernel. The eBPF verifier checks for safety properties such as program termination, safe memory access, and well-defined arithmetic operations. Once accepted by the verifier, eBPF programs are Just-in-time (JIT) compiled to the machine specific instruction set and implemented as efficiently as any other part of the kernel.

Under the hood, the eBPF verifier is designed using abstract interpretation [32]; it maintains multiple abstract domains that track eBPF register states for a given eBPF program. These abstract domains are then used to identify potential unsound behavior in the given program. It is crucial, therefore, that the analysis performed by the eBPF verifier is sound, rejecting unsafe programs and accepting only safe programs. However, logic bugs in the eBPF verifier can compromise the soundness of its analysis, leading to unsafe programs being accepted. Indeed, in recent years, the eBPF verifier has been shown to exhibit many vulnerabilities stemming from such bugs [2, 9–16, 23–26, 34, 39–41] and has also proved to be a ripe surface for attacks [50, 51, 59, 64].

Since the eBPF verifier has a direct impact on the integrity of the Linux kernel, verifying and testing the correctness of the eBPF verifier has been an ongoing effort [22, 38, 42, 46, 49, 54, 60, 65, 67, 68, 71, 73]. In our own research, we have focused on developing techniques and tools to formally reason about the value tracking performed by the verifier. Specifically, we formalized the `tnum` abstract domain, proved its soundness, and designed a sound and precise multiplication operator for the domain that has been upstreamed to the Linux kernel [20, 71]. More recently, we designed a tool, Agni [72, 73], which automatically checks the soundness of value tracking performed by the eBPF verifier by generating verification conditions for its abstract operators and proving their correctness. If no such proof can be attained, Agni generates a witness program that illustrates unsound behavior in the verifier using our differential synthesis approach [73]. Agni has since been used to patch unsound abstract operators in the verifier [4, 65]. While most of this prior work regarding the eBPF verifier pertains to its soundness, our primary focus in this paper is its precision.

***Why precision matters in the eBPF verifier?*** The eBPF verifier must ensure that its static analysis is not only sound but also sufficiently precise to allow practical use. A sound yet imprecise analysis can render the verifier ineffective by rejecting safe programs that should be accepted, simply because the analysis produces overly conservative approximations of program variables. Such imprecision arises when the verifier’s abstract domains yield coarse bounds on the possible values of program variables, especially those used in memory accesses. The verifier employs five abstract domains, four interval domains and one bitwise domain [52, 55, 61, 71], and implements a collection of efficient abstract operators for these domains. It further uses cross-domain refinements to mutually improve the precision of each domain. The overall precision of the verifier is fundamentally limited by the precision of these abstract operators themselves.

Imprecision in abstract operators can lead the eBPF verifier to reject programs that are in fact safe. Consider the following 4-bit example where a register `r1` is tracked by an unsigned interval  $[l, u]$ :

---

```

1  ...
2  r1 = r1 & 0b0011;    // mask to keep only the low two bits
3  arr[r1];             // index into a 4-byte array
4  exit;
```

---

Suppose we wish to access an offset into an array `arr` which is 4 bytes large. Here the mask guarantees that `r1`  $\in \{0, 1, 2, 3\}$ , so `arr[r1]` always stays within the 4-byte buffer and is therefore safe. For illustration purposes, let us assume the verifier’s abstract AND operator is imprecise and results in the abstract interval  $[0, 8]$ , over-approximating the set of values `r1` may take in a real execution. Unable to conclusively prove that the memory access is safe, the verifier conservatively rejects the program even though the memory access is safe and won’t go out of bounds in any execution. Similar safe rejections have prompted several recent kernel patches that improve precision of various abstract operators [1, 3, 6–8, 18].

Designing efficient yet precise operators is non-trivial. Although classical interval operators are well understood [32, 53], bit-width constraints and performance requirements complicate their direct use in the verifier without modification. Bitwise operations are particularly tricky on fixed-width interval domains: interval endpoint reasoning is unsound, while exhaustive enumeration of operand pairs is impractical. For instance, consider the bitwise AND of two unsigned intervals:  $x \in [142, 145]$ ,  $y \in [13, 15]$ . A naive abstraction might apply the AND operation only to the interval endpoints (*e.g.*,  $142 \& 13 = 12$ ,  $145 \& 15 = 1$ ), yielding the ill-formed interval  $[12, 1]$ . A less naive abstraction might try the operation on all combinations of the interval endpoints and take the minimum and maximum values, yielding the interval  $[1, 14]$ . However, enumerating all possible pairs  $(x, y)$  reveals that the actual set of outcomes is  $\{0, 1, 12, 13, 14, 15\}$ , meaning that both of these abstractions fail to capture all outputs that may result from this operation, hence they are neither sound nor precise. Importantly, an interval is not the ideal domain to represent this set since it is a sparse, non-contiguous set that cannot be tightly captured by a single interval alone. Correctly abstracting such operations requires reasoning about the bit-level structure of the operands, not just their numeric ranges. This difficulty is precisely why the eBPF verifier incorporates a dedicated bitwise domain (tnum) [71]. However, combining tnum and interval information precisely and efficiently in practice is not straight-forward and has required multiple fixes [3, 4, 6, 57, 65].

Overall, manual operator design in the verifier is labor-intensive and error-prone. Additionally, Linux developers who craft these operators generally rely on testcases to check them [5], which provides no formal guarantees of precision (*i.e.*, is the new operator implementation at least as precise as the old one for all inputs). Hence, we develop approaches to prove precision relations between two competing implementations of abstract operators in the verifier.

***Witness generation for comparing the precision of abstract operators.***

While reviewing abstract operators in the verifier, we observed that some operators may be less precise for some ranges of inputs, which causes the eBPF verifier to reject safe programs. In particular, we noticed that when the multiplication operator is given negative values as inputs, it sets its abstract domains to their widest range, essentially nullifying any useful information from the operation. Hence, we focus on developing improved operators for the eBPF verifier and develop tools and techniques to show that the new operator is more precise than the existing operator. To accomplish this goal, we need a formal framework for comparing the precision of two operator implementations. This task entails expressing the behavior of these operators in first order logic and comparing them using a logical precision specification (*i.e.*, operator A is at least as precise as operator B for all inputs). However, comparing operators across the entire space of valid abstract inputs is insufficient for evaluating real-world benefit of the new operator. Evaluating over all abstract inputs can significantly over-approximate the values that actually arise during verifier execution, many of which are unreachable by any valid eBPF program (discussed in §4.1). This phenomenon happens with the eBPF verifier because the verifier starts every register in one of two legal abstract states, *known-value* or *fully unknown*, and each subsequent instruction must transform those states through the verifier’s abstract semantics. Thus, even if a new operator is provably more precise, the gap may never manifest in real executions. To address this issue, we need to be able to generate real witness eBPF programs that illustrate precision improvements with the new operator. Our experience shows that such witnesses are rarely trivial to construct; they often cannot be hand-crafted with a single instruction or obvious inputs, but instead require multi-instruction sequences.

We propose an approach to compare the precision of two abstract operator implementations in the value tracking of the eBPF verifier. Our framework leverages our C-to-logic tool, designed in prior work [73], to express the behavior of two abstract operator implementations in logic. We develop a precision specification (§4.1) to formally prove the precision relationship between the two implementations (*e.g.*, one operator is more precise than another for some inputs). When our precision specification query shows one abstract operator is more precise for some abstract inputs, we invoke an enumerative SMT-based synthesizer (§4.2) that searches the space of bounded eBPF instruction sequences and produces a concrete eBPF program, a witness, whose analysis reaches the relevant abstract inputs. We build on Agni’s sound, but incomplete, witness generation approach [73], adapting it to precision comparisons instead of soundness violations. Using our approach, we were able to show our latest `bpf_mul` patch (§3) is more precise than the former version and we generated a witness eBPF program that illustrates this improvement. Our patch has been upstreamed to the Linux kernel [7]. We evaluate our approach on prior patches to the kernel which aim to improve precision of abstract operators in the verifier and we were able to generate witness eBPF programs for all of them, demonstrating the practical value of the precision improvements (§5). Lastly, we also use our framework to show

that the reduction operator in the verifier serves an important role in improving the precision of its abstract domains.

## 2 Background on Abstract Interpretation in the eBPF Verifier

In this section we describe how the eBPF verifier employs abstract interpretation. Specifically, we formally present how the verifier uses abstract domains and operators to perform value tracking. We then present notions of precision in the abstract interpretation literature, which will later be used to define our precision specification.

**Path-sensitive analysis without joins.** The eBPF verifier implements a path-sensitive abstract interpretation that diverges from classical join-based analyses. Instead of merging abstract states at control-flow joins using traditional join operations, it forks execution at each conditional branch and explores successor paths independently. Each execution path thus maintains its own set of abstract domains, updated by abstract operators at every instruction. This means that for every feasible execution path, the verifier maintains a separate abstract state, which allows it to retain path-specific precision that might otherwise be lost in merged (joined) states. To keep this analysis tractable, the verifier imposes strict bounds on instruction count, call stack depth, and loop unrolling.

### 2.1 Value Tracking in the eBPF Verifier

**Abstract domains for value tracking.** The eBPF verifier uses multiple abstract domains to track register values. Specifically, it uses four signed and unsigned interval domains to track the upper and lower bounds of 64-bit and 32-bit registers (*i.e.*,  $A_{u64}$ ,  $A_{u32}$ ,  $A_{s64}$ ,  $A_{s32}$ ). In addition, it uses a bitwise domain, called the `tnum` domain ( $A_{tnum}$ ) [58, 71], to track individual bits across executions. The `tnum` domain represents possible register values using a pair of 64-bit unsigned integers: a value and a mask. The value encodes known bits, while the mask specifies which bits are unknown; any bit set to 1 in the mask is unconstrained (it may be 0 or 1), whereas bits set to 0 are known and must match the corresponding bits in the value. For example, a `tnum` with value `0b10` and mask `0b01` represents the set  $\{0b10, 0b11\}$ , since the low bit is unknown and the high bit is known to be 1. This representation efficiently captures uncertainty at the bit level and is structurally similar to the bitfield domain [52].

**Signed interval domain.** The signed interval domain,  $A_{s64}$ , models the range of values a 64-bit register can take during an execution of an eBPF program, using the signed interpretation. It captures these potential values as intervals, characterized by lower and upper bounds. Formally, the domain is defined as follows:

$$A_{s64} \triangleq \{[l, u] \mid l, u \in \mathbb{Z}_{64}, l \leq_{s64} u\} \cup \{\perp\}.$$

Here,  $\mathbb{Z}_{64}$  denotes the set of all 64-bit signed integers,  $\leq_{s64}$  represents signed integer comparison for 64-bit integers, and  $\perp$  represents the empty set. The C implementation within the eBPF verifier maintains these intervals using two signed 64-bit integers, `s64_min` and `s64_max`, which indicate the current minimum and maximum possible values for the register. The *concretization function*  $\gamma_{s64}$  translates the signed interval domain into the set of concrete values it represents, in this case:

$$\gamma_{s64}(a) \triangleq \begin{cases} \{z \in \mathbb{Z}_{64} \mid l \leq_{s64} z \leq_{s64} u\} & \text{if } a = [l, u] \\ \emptyset & \text{if } a = \perp \end{cases}$$

Conversely, the *abstraction function*  $\alpha_{s64}$  converts a concrete set of values  $c \subseteq \mathcal{P}(\mathbb{Z}_{64})$  into the interval containing all elements of  $c$ :

$$\alpha_{s64}(c) \triangleq \begin{cases} [\min_{s64}(c), \max_{s64}(c)] & \text{if } c \neq \emptyset \\ \perp & \text{if } c = \emptyset \end{cases}$$

where  $\min_{s64}(c)$  and  $\max_{s64}(c)$  calculate the minimum and maximum of the finite set  $c$ , respectively.

An *abstract operator* is the abstract interpretation counterpart of a concrete operation. Consider the subtraction abstract operator for this domain [31]. Let the subtraction operator  $\ominus_{s64}$  on signed intervals be defined as follows; given two intervals  $a_1 = [l_1, u_1]$  and  $a_2 = [l_2, u_2]$  in the signed 64-bit integer domain, the abstract subtraction computes a resulting interval that approximates all possible concrete subtractions between elements from these intervals:

$$a_1 \ominus_{s64} a_2 \triangleq \begin{cases} [l_1 -_{s64} u_2, u_1 -_{s64} l_2] & \text{if no overflow} \\ [-2^{63}, 2^{63} - 1] & \text{otherwise} \end{cases}$$

Here, overflow is checked according to signed 64-bit arithmetic. If overflow occurs, the operation yields the maximal interval, ensuring soundness at the expense of precision. The eBPF verifier implements this abstract operator as defined above. Importantly, the other interval domains in the verifier follow the same formalism as the signed domain, adjusted according to their respective bit-width and signedness. Specifically, for the unsigned interval domains (e.g.,  $A_{u64}$ ), intervals similarly represent ranges of values using lower and upper bounds, but with arithmetic and comparisons interpreted according to unsigned integer semantics. Consequently, their minimal bound is always non-negative, and arithmetic overflow is checked using the corresponding unsigned arithmetic rules.

**The reduction operator in the eBPF verifier.** The eBPF verifier does not combine information from multiple domains using traditional approaches, such as Cartesian or modular reduced products [33]. Instead, the verifier mixes abstraction and reduction [65, 73] to improve precision of its abstract domains. For example, when computing interval results for bitwise operations (e.g., `r1 = r1 & 0b10`), the verifier does not separately calculate interval and tnum domain

values and then combine them via a formal product. Instead, it computes the abstract value in the tnum domain first (*e.g.*, determining bits known to be fixed and unknown), and subsequently uses this information to derive an interval bound.

Beyond this strategy, the verifier also maintains a reduction operator which is shared by all abstract operators in the verifier [48] and is performed immediately after every abstract operator (arithmetic, logic, branch) is executed. The purpose of this operator is to systematically propagate information between domains to improve precision. Let each abstract domain  $(A_i, \sqsubseteq_i)$  be equipped with an abstraction function  $\alpha_i : \mathcal{P}(C_i) \rightarrow A_i$  and a concretization function  $\gamma_i : A_i \rightarrow \mathcal{P}(C_i)$  defined over the same concrete domain  $\mathcal{P}(C_i)$ , where  $C_i$  denotes the set of concrete values associated with the  $i^{\text{th}}$  abstract domain (*e.g.*, signed 64-bit integers for  $A_{s64}$ , unsigned 64-bit integers for  $A_{u64}$ , etc.). Here,  $\sqsubseteq_i$  denotes the partial order on the abstract domain  $A_i$ . The combined abstract domain is then the product lattice  $A = A_1 \times \dots \times A_k$  ordered component-wise:

$$(a_1, \dots, a_k) \sqsubseteq (b_1, \dots, b_k) \iff \forall i, a_i \sqsubseteq_i b_i.$$

The concretization of a product domain is commonly defined as the intersection of its component concretizations:  $\gamma(a) = \bigcap_{i=1}^k \gamma_i(a_i)$ . However, such a definition would be unsound in the context of the eBPF verifier's product domain; intersecting the concretizations of signed with unsigned interval domains would erroneously eliminate valid negative values, and intersecting concretizations of 32-bit and 64-bit domains would constrain values to 32-bit ranges. To avoid these issues, the verifier does not use an intersection-based concretization. Instead, it maintains domain-specific concretizations and refines them through its reduction operator. We define the product domain concretization  $\gamma^*$  as the tuple of its component concretizations:

$$\gamma^*(a) := (\gamma_1(a_1), \dots, \gamma_k(a_k)), \quad \text{with each } \gamma_i(a_i) \subseteq C_i.$$

We write  $\subseteq^*$ ,  $\subset^*$ ,  $\supseteq^*$ , and  $\supset^*$  to denote component-wise set inclusion between concretization tuples. For example,  $\gamma^*(a) \subseteq^* \gamma^*(b) \iff \forall i, \gamma_i(a_i) \subseteq \gamma_i(b_i)$ . Formally, the verifier's reduction operator is defined as a monotone function  $\rho : A \rightarrow A$ . Given an abstract state  $a = (a_1, \dots, a_k) \in A$ , the reduction operator computes a refined abstract state  $a' = \rho(a)$  satisfying

$$a' \sqsubseteq a \quad \text{and} \quad \gamma^*(a') \subseteq^* \gamma^*(a),$$

thereby locally refining each domain without enforcing an explicit intersection. The verifier strikes a balance between precision and performance by applying the reduction operator only a bounded number of times (usually two iterations), following a partially reduced products approach [53]. Informally, each invocation of the reduction operator consists of the following three sequential reduction steps:

1. **Bitwise-to-interval refinement:** Use known and unknown bits from the tnum domain to restrict possible values in each 64-bit signed and unsigned interval.

2. **Interval cross-domain propagation:** Propagate tightened bounds between 64-bit and 32-bit intervals, and between signed and unsigned intervals, ensuring consistency across width and signedness.
3. **Interval-to-bitwise refinement:** Use the updated intervals to detect newly fixed bits and mark them as known within the tnum domain.

Initially, the reduction operator was introduced without a formal soundness proof into the eBPF verifier. We proved the soundness of the verifier’s reduction operator in prior work [65]. In this paper, we focus on its role in enhancing precision. In Sec. §5, we empirically evaluate it by comparing abstract operator precision with and without the reduction step.

## 2.2 Comparing the Precision of Abstract Operators

The eBPF verifier utilizes five non-relational abstract domains. We compare two abstract operators by ordering their results in the underlying abstract lattice, following the standard framework of abstract interpretation [32, 53]. To compare the precision of two abstract operators, we assume sound abstractions, which we define next. Let  $f : C \rightarrow C$  be a concrete operation over the set of all 64-bit machine values, and let  $f^* : \mathcal{P}(C) \rightarrow \mathcal{P}(C)$  denote its collecting semantics. An abstract operator  $F^\# : A \rightarrow A$  is *sound* with respect to  $f$  if, for every abstract state  $a = (a_1, \dots, a_k) \in A$  and for each component  $i$ , the following holds:

$$f^*(\gamma_i(a_i)) \subseteq \gamma_i(\pi_i(F^\#(a))),$$

where  $\pi_i : A \rightarrow A_i$  denotes the projection onto the  $i^{\text{th}}$  component of the product lattice  $A$ . This definition ensures that soundness is preserved individually within each component domain.

**Definition 1 (Precision preorder, multi-domain).** *Given two sound abstract operators  $F_1^\#, F_2^\# : A \rightarrow A$ , we say  $F_1^\#$  is at least as precise as  $F_2^\#$ , denoted  $F_1^\# \preceq F_2^\#$ , if and only if:*

$$\forall a \in A : F_1^\#(a) \sqsubseteq F_2^\#(a)$$

where  $\sqsubseteq$  is the component-wise product order.

Intuitively, operator  $F_1^\#$  is at least as precise as  $F_2^\#$  if, for all inputs, the abstract states produced by  $F_1^\#$  are component-wise at least as informative (*i.e.*, smaller or equal in the lattice ordering) than those produced by  $F_2^\#$ . Equivalently, this implies  $\gamma^*(F_1^\#(a)) \subseteq^* \gamma^*(F_2^\#(a))$  for every  $a \in A$ . If the opposite direction also holds ( $F_2^\# \preceq F_1^\#$ ), then  $F_1^\#$  and  $F_2^\#$  are equally precise. If neither direction holds universally, the operators are incomparable.

## 3 Improving the Precision of Abstract Operators in the eBPF Verifier

While exploring abstract operators in the eBPF verifier, we observed that some abstract operators perform overly conservative approximations of their concrete



counterparts, which resulted in loss of precision. Specifically, the eBPF verifier's multiplication operator seemed to exhibit this exact behavior (*i.e.*, returning loose bounds when given negative operands). To tackle this imprecision, we first needed to formalize how the verifier performs this operator to identify where imprecision stems from. Then, we wanted to craft a new operator that is at least as precise as the old operator for all abstract inputs, but also more precise for some inputs. This section presents these efforts. We first describe how the abstract operator for multiplication (`bpf_mul`) is performed by the eBPF verifier and where imprecision stems from. Then we propose an improved multiplication abstract operator and discuss proving its precision merits over the prior operator.

Beyond serving as a stand-alone improvement, this case study motivates the need for a more systematic approach to design and validate abstract operators. Crafting a more precise operator is only part of the challenge, ensuring that this improved precision is relevant to real-world eBPF programs is equally important. In particular, reasoning about all abstract inputs can over-approximate what is actually reachable in verifier executions, since the verifier begins analysis from constrained initial states and enforces strict invariants on how abstract states evolve. As such, many abstract inputs used in theoretical comparisons may never arise in practice. We present our techniques for comparing precision and witness generation in the following sections (§4).

**How the eBPF verifier performs abstract multiplication (`bpf_mul`).** Formally, we represent the existing multiplication operator in the eBPF verifier as  $F_{\text{ebpf\_mul}}^\# : A \times A \rightarrow A$  where  $A = A_{u64} \times A_{s64}$ . While the verifier uses five abstract domains, we focus on these two domains for clarity and brevity as these two domains are enough to illustrate how imprecision manifests in this operator.

*Operands.* We define the operator's operands  $a, b \in A$

$$a = ([\ell_{u64}^a, u_{u64}^a], [\ell_{s64}^a, u_{s64}^a]), \quad b = ([\ell_{u64}^b, u_{u64}^b], [\ell_{s64}^b, u_{s64}^b]),$$

where unsigned and signed interval invariants are maintained,  
 $0 \leq \ell_{u64}^{(\cdot)} \leq u_{u64}^{(\cdot)} \leq 2^{64} - 1$  and  $-2^{63} \leq \ell_{s64}^{(\cdot)} \leq u_{s64}^{(\cdot)} \leq 2^{63} - 1$ .

*Bit-width constants.* The following constants represent the minimum and maximum numeric values that can be represented using a fixed number of bits, as used by the verifier, when interpreted as signed or unsigned integers.

$$\begin{aligned} U64_{\min} &= 0, & U64_{\max} &= 2^{64} - 1, \\ S64_{\min} &= -2^{63}, & S64_{\max} &= 2^{63} - 1, \\ U32_{\max} &= 2^{32} - 1. \end{aligned}$$

*Operator definition as represented by the eBPF verifier.*

$F_{\text{ebpf\_mul}}^\#(a, b) = \begin{cases} \top, & \text{if } \ell_{s64}^a < 0 \vee \ell_{s64}^b < 0, \\ \top, & \text{if } u_{u64}^a > U32_{\max} \vee u_{u64}^b > U32_{\max}, \\ ([\ell_{u64}^a \cdot \ell_{u64}^b, u_{u64}^a \cdot u_{u64}^b], \Delta), & \text{otherwise} \end{cases}$
--

where  $\top := ([U64_{\min}, U64_{\max}], [S64_{\min}, S64_{\max}])$  is the greatest element of  $A$  and

$$\Delta = \begin{cases} [S64_{\min}, S64_{\max}], & u_{u64}^a \cdot u_{u64}^b > S64_{\max} \\ [\ell_{u64}^a \cdot \ell_{u64}^b, u_{u64}^a \cdot u_{u64}^b], & \text{otherwise.} \end{cases}$$

$F_{\text{ebpf-mul}}^\#$  computes precise interval products under restrictive conditions; both operands must be non-negative and fit within 32-bit unsigned bounds, and the resulting product must not exceed the maximum representable 64-bit signed value,  $S64_{\max}$ . Every other case is conservatively widened to the full 64-bit range. While this ensures soundness, it significantly limits the utility of the operator in practice, especially for signed computations, leading to overly conservative analyses and potential rejection of safe programs. As shown in Table 1, this behavior results in avoidable precision losses. To address this issue, we present an improved multiplication operator that retains soundness while computing tighter interval bounds in many of these previously imprecise cases.

**Our more precise eBPF multiplication abstract operator.** Our operator is grounded in interval arithmetic [56], which has become standard in abstract interpretation literature [32, 53]. Specifically, we adapt interval multiplication to 64-bit fixed-width arithmetic, accounting for overflow as defined by the eBPF instruction set [45]. We formally define our abstract multiplication operator  $F_{\text{our-mul}}^\# : A \times A \rightarrow A$  where  $A = A_{u64} \times A_{s64}$ . We use these two domains for brevity, but the operator is also directly extensible to the  $A_{u32}$  and  $A_{s32}$  domains. Our operator preserves the existing abstract multiplication for the tnum domain defined elsewhere [71]. Next, we define auxiliary products which we use to compute intervals:

$$\begin{aligned} U_{\ell\ell} &= \ell_{u64}^a \cdot \ell_{u64}^b, & U_{uu} &= u_{u64}^a \cdot u_{u64}^b, \\ S_{\ell\ell} &= \ell_{s64}^a \cdot \ell_{s64}^b, & S_{\ell u} &= \ell_{s64}^a \cdot u_{s64}^b, \\ S_{u\ell} &= u_{s64}^a \cdot \ell_{s64}^b, & S_{uu} &= u_{s64}^a \cdot u_{s64}^b. \end{aligned}$$

Lastly, let  $prod_{\min} = \min\{S_{\ell\ell}, S_{\ell u}, S_{u\ell}, S_{uu}\}$  and  $prod_{\max} = \max\{S_{\ell\ell}, S_{\ell u}, S_{u\ell}, S_{uu}\}$ . Our improved multiplication operator is defined as follows:

$$\boxed{F_{\text{our-mul}}^\#(a, b) = (\Delta_{u64}, \Delta_{s64})}$$

where

$$\begin{aligned} \Delta_{u64} &= \begin{cases} [U64_{\min}, U64_{\max}], & \text{overflow in } U_{\ell\ell} \text{ or } U_{uu} \\ [U_{\ell\ell}, U_{uu}], & \text{otherwise} \end{cases} \\ \Delta_{s64} &= \begin{cases} [S64_{\min}, S64_{\max}], & \text{overflow in } S_{\ell\ell} \text{ or } S_{\ell u} \text{ or } S_{u\ell} \text{ or } S_{uu} \\ [prod_{\min}, prod_{\max}], & \text{otherwise.} \end{cases} \end{aligned}$$

Our operator multiplies the operand intervals' bounds, returning precise unsigned and signed 64-bit product intervals when all auxiliary products do not

Table 1: Precision gains in our abstract multiplication operator ( $F_{\text{our-mul}}^\#$ ) compared to the verifier’s operator ( $F_{\text{ebpf-mul}}^\#$ ) where  $a, b \in A_{u64} \times A_{s64}$ . For instance, when any of the operands’ signed intervals maintain negative values (second row), the verifier’s operator will set its abstract values to  $\top$ , whereas ours will produce tight bounds when no overflow is possible.

Operands $a, b$	$F_{\text{ebpf-mul}}^\#(a, b)$	$F_{\text{our-mul}}^\#(a, b)$
$([2, 3], [2, 3]), ([4, 5], [4, 5])$	$([8, 15], [8, 15])$	$([8, 15], [8, 15])$
$([1, 1], [1, 1]), ([0, 0], [-1, -1])$	$\top$	$([0, 0], [-1, -1])$
$([2^{34}, 2^{34}], [2^{34}, 2^{34}]), ([1, 1], [1, 1])$	$\top$	$([2^{34}, 2^{34}], [2^{34}, 2^{34}])$
$([0, 5], [0, 5]), ([0, 2], [-2, 2])$	$\top$	$([0, 10], [-10, 10])$
$([2^{63}, 2^{63}], [0, 2^{62}]), ([2, 8], [2, 8])$	$\top$	$\top$

overflow. If any product overflows, the operator conservatively widens the affected domain to the full 64-bit range. Table 1 exemplifies precision gains of our operator over the existing one.

**Bridging formal precision and real world execution—why we need automated precision comparison.** The construction above guarantees that our operator  $F_{\text{our-mul}}^\#$  is *never less precise* than the kernel’s current operator  $F_{\text{ebpf-mul}}^\#$ , but we don’t know if it is practically more precise in real eBPF executions since abstract inputs for which our operator is more precise may not be reachable under the verifier’s strict constraints. Further, most of the verifier’s operators are handcrafted by kernel developers and may involve subtle bit-level invariants, or interaction with multiple abstract domains in non-obvious ways (*i.e.*, bitwise operations for signed intervals). This means that we cannot rely on intuition or isolated examples to assess their precision. Hence, when a new abstract operator is proposed, in our case the multiplication operator, we argue it should achieve two goals: **(i)** *satisfy* the precision preorder  $F_{\text{our-mul}}^\# \preceq F_{\text{ebpf-mul}}^\#$  for *all* abstract inputs (or a region of inputs that are of interest) and **(ii)** *demonstrate* concrete eBPF programs in which that improved precision is observable (*i.e.*, in the multiplication case, programs for which the old operator widens to  $\top$  while the new one produces a tight bound).

Task (i) reduces to a lattice-theoretic proof obligation. Task (ii) amounts to searching the program space for witnesses where precision differences manifest. Performing this search by hand is error-prone and does not scale to the many arithmetic and branch operators maintained in the verifier. Hence we propose an *automated* pipeline that (a) encodes our precision specification as a logical formula, (b) systematically enumerates bounded eBPF instruction sequences, and (c) invokes an SMT solver to find a satisfying assignment that separates the two operators in terms of precision. The next section introduces our synthesis-driven framework for precision comparison.

## 4 Our Approach for Precision Comparison and Witness Synthesis

In this section, we describe how precision relationships between abstract operators can be formalized and checked via logical queries. To ground these formal guarantees in practice, we develop an automated pipeline that synthesizes concrete eBPF programs exposing true precision differences in the verifier. This two-part approach, proving precision formally and validating it empirically, ensures that abstract operator changes are both correct and meaningful in the context of real verifier executions.

### 4.1 Precision Specification for Comparing Abstract Operators

Let  $F_1^\#, F_2^\# : A \times A \rightarrow A$  be two *sound* abstract operators over the domain  $A = A_{u64} \times A_{u32} \times A_{s64} \times A_{s32} \times A_{tnum}$ . To show that  $F_1^\# \preceq F_2^\#$ , we check the validity of the following query:

$$\begin{aligned} & \forall t, u \in C, \quad a_t, a_u \in A : \\ & mem_A(t, a_t) \wedge mem_A(u, a_u) \wedge \\ & a_v = F_1^\#(a_t, a_u) \wedge a'_v = F_2^\#(a_t, a_u) \implies a_v \sqsubseteq a'_v. \end{aligned} \quad (1)$$

Here,  $mem_A(x, a) \triangleq \bigwedge_{i=1}^k x \in \gamma_i(\pi_i(a))$  denotes that a concrete value  $x$  is a member of the abstract state  $a \in A$  iff it lies in the concretization of every component of  $a$ . We use  $a_v = F_1^\#(a_t, a_u)$  to represent an abstract operator and its input-output relationships as specified by the verifier's source code. Specifically,  $a_t$  and  $a_u$  represent the inputs to  $F_1^\#$  and  $a_v$  the output. With this query, we compare each component (*i.e.*,  $A_{s64}$ ,  $A_{u64}$ , etc.) of the abstract states produced by both abstract operators  $F_1^\#$  and  $F_2^\#$  and assert the precision relationship between them. For interval domains, checking whether  $a_v \sqsubseteq a'_v$  amounts to verifying that  $\ell' \leq \ell \wedge u \leq u'$ , where  $[\ell', u']$  and  $[\ell, u]$  are the intervals represented by  $a'_v$  and  $a_v$ , respectively. For the tnum domain, where abstract values are represented as value-mask pairs  $(av', am')$  and  $(av, am)$ , we can ascertain  $a_v \sqsubseteq a'_v$  by checking  $((av \ \& \ \sim am') = av') \wedge (am \mid am') = am'$ .

We are also able to check the precision of individual components between the two operators. Let's assume we proved  $F_1^\# \preceq F_2^\#$  for two operators we are comparing and we suspect that the  $A_{s64}$  component produced by  $F_1^\#$  is not only at least as precise as the one produced by  $F_2^\#$  for all inputs, but also more precise for some inputs. Using the previously defined projection function  $\pi_i : A \rightarrow A_i$ , which extracts the  $i^{\text{th}}$  component of an abstract state, we express this query formally as:

$$\begin{aligned} & \exists t, u \in C, \quad a_t, a_u \in A : \\ & mem_A(t, a_t) \wedge mem_A(u, a_u) \wedge \\ & a_v = F_1^\#(a_t, a_u) \wedge a'_v = F_2^\#(a_t, a_u) \implies \pi_{s64}(a_v) \sqsubset \pi_{s64}(a'_v). \end{aligned} \quad (2)$$

Table 2: Reachable and unreachable abstract states where  $a \in A_{s64} \times A_{u64}$ . R1 exhibits a case where both abstract values are aligned after reduction, representing the same values in their respective domains. R2 represents a reachable state where the signed interval cannot be used to inform the unsigned interval since  $[-100, 200]$  cannot be represented soundly in an unsigned interval. U1 and U2 exemplify abstract states that have not been reduced which cannot happen in a real verifier execution, hence they are unreachable. After the reduction operation, U1 and U2 should be  $([-10, -1], [2^{64} - 10, 2^{64} - 1])$  and  $([5, 10], [5, 10])$ , respectively.

Case	$a$	Reachable?
R1	$([-5, -1], [2^{64} - 5, 2^{64} - 1])$	✓
R2	$([-100, 200], [0, 2^{64} - 1])$	✓
U1	$([-10, -1], [0, 2^{64} - 1])$	×
U2	$([5, 2^{32}], [5, 10])$	×

We can adapt this query to reason about any of the other specific domains as well (*i.e.*,  $A_{s32}$ ,  $A_{tnum}$ , etc.). Reasoning about precision of specific domains is useful because patches to the verifier’s abstract operators may only improve the precision of a specific domain rather than all.

**Reachability constraints in the verifier’s abstract state space.** While proving any of the above queries is useful for understanding the precision relations between two comparable operators, it may not reflect real-world precision improvements in the verifier. This is because the abstract states used in such specifications over-approximate the set of states that can arise during actual eBPF program analysis. In practice, the verifier initializes registers in either known or fully unknown states, and these states evolve only through a sequence of eBPF abstract operations and reduced by the reduction operator. As such, many abstract states used in the precision specification may never actually be realized during verifier execution. Table 2 illustrates examples of reachable and unreachable abstract states in the verifier, using only the 64-bit signed and unsigned domains for brevity. Consider the last row in the table; while it is a valid abstract state where each individual abstract value is sound, it is not reachable in a real verifier execution. This is because the reduction operator would tighten the signed interval such that the resulting state would become  $([5, 10], [5, 10])$ . Importantly, our precision specification considers all of these abstract states when comparing operators, regardless of reachability. If any unreachable states are used as inputs in our query to show precision improvements of one abstract operator over another, then that precision gain is impractical and no real eBPF program can illustrate it. Since we are interested in practical precision improvements that can manifest in a real verifier execution, we now present our witness generation approach for synthesizing eBPF programs that illustrates reachable precision improvements in one abstract operator over another.

## 4.2 Synthesizing Witness eBPF Programs for Precision Comparisons

Proving precision properties of abstract operators using our precision specification (§4.1) may not be enough to determine if one operator is truly more precise than another in any real eBPF program. This discrepancy underscores the need to test precision differences not just in theory, but through actual witness programs that drive the verifier into states where the two operators diverge. Our goal then is to automatically synthesize actual eBPF programs that illustrate an instance in which one abstract operator produces tighter bounds in one, or more, of its abstract domains compared to another operator as specified in Eqn. 2.

To address this challenge, we adapt our differential synthesis approach [73], originally developed for soundness verification, to instead generate witness programs that highlight precision differences between abstract operators. We use `bpf_add` as our abstract operator under test to illustrate our witness generation in this section. Concretely, our approach aims to model the verifier’s behavior when executed with two competing abstract operators. (*i.e.*, two different operators for `bpf_add`). This involves enumerating bounded length eBPF programs that exercise the abstract operators of interest. We use an SMT solver to determine when a program illustrates that one operator is more precise than the other. This method is sound—but incomplete; failure to find a witness at a given bound does not preclude its existence.

**Reaching abstract states starting from initial states.** The goal of our synthesis procedure is to produce an abstract state  $a \in A$  which is the result of the verifier’s analysis of a sequence of eBPF instructions. Importantly, the verifier begins all executions with an initial set of abstract states it allows. Formally, we use  $init(b)$  to specify that abstract input  $b \in A$  is an initial abstract state where  $b$  can be unknown ( $b = \top$ ) or a singleton (*i.e.*,  $b_{s64} = [\ell, \ell]$  where  $\ell \in \mathbb{Z}_{64}$ ). Hence,  $a$  is reachable if there exists a sequence of eBPF instructions for which the verifier’s analysis reaches  $a$  starting from the set of restricted abstract states such that  $init(b)$  holds.

**Witness generation procedure.** Let  $F_1^\#$  and  $F_2^\#$  be two sound, comparable abstract operators that satisfy our precision specification (Eqns. 1 and 2). Our goal is to construct an executable eBPF program whose analysis in the verifier reaches an abstract state where the two operators diverge in precision. To generate a concrete witness program, we consider all instruction sequences up to a maximum length  $L$ . Each sequence includes a mix of arithmetic, logic, and branch instructions, with the final instruction reserved for the abstract operator under test. These programs are partially specified: instruction opcodes are fixed, but operands and data flow remain symbolic. For each such program, we construct a logical formula that (i) selects initial abstract states consistent with the verifier’s start-state constraints (*e.g.*, known or fully unknown abstract states) using  $init()$ , (ii) models each instruction according to the verifier’s abstract semantics, and (iii) ensures that the final instruction produces a strictly

more precise abstract state under one abstract operator than the other. We emit every such program formula to an SMT solver and repeat this process until we reach our maximum bound length  $L$  or time limit.

To illustrate how this process begins, we first consider the case where the program consists of a single instruction, the abstract operator under test. In this case, we check whether the precision gap between  $F_{old\_add}^\#$  and  $F_{new\_add}^\#$  (modeling `bpf_add`) can be demonstrated using only initial abstract states. For illustration purposes, we test for precision differences in the  $A_{s64}$  domain, but this applies to any domain. This reduces to a satisfiability query over concrete and abstract inputs:

$$\begin{aligned} & t, u \in C, \quad a_t, a_u \in A : \\ & init(a_t) \wedge init(a_u) \wedge mem_A(t, a_t) \wedge mem_A(u, a_u) \wedge \\ & a_v = F_{new\_add}^\#(a_t, a_u) \wedge a'_v = F_{old\_add}^\#(a_t, a_u) \wedge \pi_{s64}(a_v) \sqsubset \pi_{s64}(a'_v) \end{aligned} \quad (3)$$

**Extending witness generation to multiple instructions.** A single instruction may not be enough to elicit precision gaps between operators so we extend our synthesis to a larger program length. We explore all instruction sequences of length  $L - 1$  and reserve the  $L^{\text{th}}$  instruction for the operator under test. Operands are left symbolic. We assert that every operand must be either (i) an initial abstract state allowed by the verifier or (ii) the output produced by an earlier instruction. Extending Eqn. 3, we illustrate a two-instruction (`bpf_and` followed by `bpf_add`) query:

$$\begin{aligned} & p, q, r, t, u, v \in C, \quad a_p, a_q, a_r, a_t, a_u, a_v \in A : \\ & init(a_p) \wedge init(a_q) \wedge mem_A(p, a_p) \wedge mem_A(q, a_q) \wedge \\ & r = conc_{and}(p, q) \wedge a_r = F_{and}^\#(a_p, a_q) \wedge mem_A(r, a_r) \wedge \\ & (init(a_t) \vee assign(t, \{p, q, r\})) \wedge (init(a_u) \vee assign(u, \{p, q, r\})) \wedge \\ & mem_A(t, a_t) \wedge mem_A(u, a_u) \wedge \\ & a_v = F_{new\_add}^\#(a_t, a_u) \wedge a'_v = F_{old\_add}^\#(a_t, a_u) \wedge \pi_{s64}(a_v) \sqsubset \pi_{s64}(a'_v). \end{aligned} \quad (4)$$

Here, variables  $p, q, \dots, v \in C$  denote concrete 64-bit values, and  $a_p, a_q, \dots, a_v \in A$  are their abstract counterparts. For the first instruction `bpf_and`,  $r = conc_{and}(p, q)$  represents the concrete operator and  $a_r = F_{and}^\#(a_p, a_q)$  represents its abstract counterpart. The abstract operator consumes initial abstract inputs  $a_p, a_q$ ; its result  $a_r$  may later be used as an operand. For the second instruction, `bpf_add`, each abstract input may be *either* a fresh initial element  $init(\cdot)$  *or* one of the earlier results  $\{a_p, a_q, a_r\}$ . We encode this choice with  $assign(x, \{y_1, \dots, y_m\}) = \bigvee_{i=1}^m (x = y_i \wedge a_x = a_{y_i})$ , thereby linking concrete values and abstract states. Finally, the key precision-checking constraint is imposed:  $\pi_{s64}(a_v) \sqsubset \pi_{s64}(a'_v)$ , asserting that the new operator's output interval is strictly more precise than that of the old operator. If this formula is satisfiable, we get a model of a concrete two-instruction eBPF program which exposes a real precision advantage of the new operator over the old one. Otherwise we keep exploring the search space.

## 5 Experimental Evaluation

In this section, we evaluate our prototype which compares the precision of two abstract operator implementations in the eBPF verifier and produces witness eBPF programs to illustrate precision gaps between these operators in real eBPF programs. We test our framework on recent patches to the kernel that aim to improve precision of various abstract operators in range tracking. Additionally, we use our approach to evaluate the effectiveness of the verifier’s reduction operator (called `BPF_SYNC`) which is called after every abstract operator executes. For brevity, we compare a limited set of arithmetic and logical abstract operators with and without reduction and present the results. Overall, with this evaluation we tackle the following questions:

1. **RQ1: Precision Comparison.** Can our formal framework determine whether a newly proposed abstract operator implementation is at least as precise, or strictly more precise, compared to an existing implementation, particularly in the kernel’s precision-related patches?
2. **RQ2: Witness Generation.** Given identified precision gaps between operator implementations, can our automated framework reliably generate real eBPF programs as concrete witnesses that reveal these differences during verifier execution?
3. **RQ3: Impact of Reduction Operator.** Does the eBPF verifier’s reduction operator (`BPF_SYNC`) tangibly improve the precision of abstract domains, and can our approach demonstrate these improvements through synthesized witness programs?

**Experimental setup.** We conducted all experiments on a system running Ubuntu 20.04, equipped with an AMD Ryzen 5 3600 (6-core CPU) and 32GB RAM. Our prototype was implemented in Python 3.12, leveraging the Agni C-to-logic translation tool [73] to automatically convert abstract operator implementations from C into logical representations. For logical verification and witness synthesis tasks, we utilized the Z3 SMT solver [35], setting a query timeout of 10 minutes and synthesizing eBPF programs up to a length of 4 instructions.

**Evaluating kernel precision patches (RQ1 & RQ2).** To evaluate the effectiveness of recent precision improvement efforts, we applied our precision comparison and witness generation framework to a set of kernel patches spanning Linux versions v5.7 to v6.8. These patches targeted various abstract operators, including the reduction operator (`BPF_SYNC`), to address overly conservative behavior in the verifier that resulted in rejection of safe programs. Table 3 presents a summary of our evaluation. As shown in columns 2 and 3, our framework confirmed that all patched operators are *at least as precise* as their predecessors across all inputs and domains, and *strictly more precise* for some. Column 4 details the specific abstract domains where each patch demonstrably improved precision, ranging from individual domains like  $A_{s64}$  to improvements across all tracked domains. To assess the practical benefit of these precision improvements, we used our synthesis engine to generate concrete eBPF witness programs



Table 3: We show precision relation and witness generation results for selected eBPF abstract operator patches (Linux v5.7–v6.8). For each modified eBPF instruction we check the two-way precision preorder between the new and old abstract operators and indicate the abstract domains in which the patch demonstrably tightens results. We are able to generate real eBPF program as witnesses for all patches that are provably more precise for some inputs.

eBPF insn	Patch reference	$F_{\text{new}}^{\#} \preceq F_{\text{old}}^{\#}$	$F_{\text{old}}^{\#} \preceq F_{\text{new}}^{\#}$	Improved domains	Witness generated?
BPF_AND	[8]	✓	×	$A_{s64}, A_{s32}$	✓
BPF_SYNC	[57]	✓	×	<i>All</i>	✓
BPF_SYNC	[6]	✓	×	$A_{tnum}$	✓
BPF_XOR	[1]	✓	×	<i>All</i>	✓
BPF_MUL	[7]	✓	×	$A_{u64}, A_{u32},$ $A_{s64}, A_{s32}$	✓

for each precision gain (column 5), providing evidence that these improvements are observable in a real execution. These synthesized programs complement the hand-written examples that often accompany kernel patches, reinforcing the role of automated witness generation in kernel development and validation.

```

1  set r1          ; r1 unknown r1([S64min, S64max], [S32min, S32max])
2  r1 = r1 & 15    ; r1([0, 15], [0, 15])
3  r1 = r1 - 10    ; r1([-10, 5], [-10, 5])
4  r1 = r1 * -5    ; our bpf_mul: r1([-75, 50], [-75, 50])
5                  old bpf_mul: r1([S64min, S64max], [S32min, S32max])
6  exit

```

Listing 1.1: Witness eBPF program demonstrating improved precision in the multiplication abstract operator `bpf_mul` for signed interval domains ( $A_{s64}, A_{s32}$ ). The old operator yields overly conservative intervals, whereas our operator computes exact bounds.

**Generating witness of precision for `bpf_mul` and `bpf_and`.** Here, we present two witness programs generated by our synthesizer for our improved multiplication abstract operator (§3) and for the bitwise *and* abstract operator `bpf_and` [8]. These witnesses are real eBPF programs that exhibits the precision merits of these operators. We illustrate our multiplication operator’s performance on signed values in Listing 1.1 using two domains,  $A_{s64}$  and  $A_{s32}$ . The program exemplifies how the operator handles multiplying two signed intervals  $[-10, 15] \times [-5, -5]$ . We observe that the old multiplication operator (line 5) yields the widest range possible for these domains while our operator yields the exact result expected for this multiplication  $[-75, 50]$ . This improvement in precision could practically mean the verifier would accept a program in which the `r1`

register might be later used to access memory. Importantly, in our patch to the Linux kernel, we included such an automatically generated witness embedded in its commit message. This patch is now upstreamed to kernel version v6.14 and beyond [7].

Our second witness example (Listing 1.2) illustrates precision improvement in the  $A_{s64}$  domain for the `bpf_and` operator. The patch itself aimed at improving the handling of negative values when performing the bitwise *and* operation, since it would result in verifier rejections detailed in [8]. Our synthesizer generates a program which performs bitwise *and* on two signed intervals where the old operator (line 5) results in  $\top$  and the proposed version (line 4) returns a more precise interval  $[-2^{40}, \approx 2^{40}]$ .

---

```

1  set r1                ; set r1 to unknown r1([S64min, S64max])
2  set r2                ; set r2 to unknown r2([S64min, S64max])
3  r1 = r1 s>> 23        ; r1([-1099511627776, 1099511627775])
4  r2 = r2 s>> 32        ; r2([-2147483648, 2147483647])
5  r1 = r1 & r2          ; new bpf_and: r1([-1099511627776, 1099511627775])
6                        ; old bpf_and: r1([S64min, S64max])
7  exit

```

---

Listing 1.2: Witness eBPF program illustrating precision gap in abstract operator `bpf_and` for the signed interval domain ( $A_{s64}$ ).

**Evaluating the reduction operator (RQ3).** To assess the precision benefits of the verifier’s reduction operator (`BPF_SYNC`), we systematically compared a set of arithmetic and logical abstract operators with and without reduction, using Linux kernel version v6.10. The reduction operator, invoked at the tail end of every abstract operator execution, has been the focus of multiple precision- focused kernel patches [3, 6, 18]. While its soundness has been formally established [65], its precision impact had not been rigorously evaluated prior to this work.

Table 4 summarizes our findings and reveals several important insights. First, as shown in columns 2 and 3, all operators augmented with reduction are always at least as precise, and some may also be strictly more precise, than operators without reduction. Second, column 4 illustrates that the reduction operator can enhance precision across a wide range of domains, frequently yielding improvements in all five abstract domains. Third, our framework was able to successfully synthesize witness eBPF programs (column 5) for all operators and for most domains where precision gains were possible. However, we were not able to generate witnesses that expose improvements in the `tnum` domain for any of the bitwise/shift operations given the program length and time constraints used in this experiment. This reflects the inherent limitations of bounded enumeration; we cannot know if limited sequence length prevented reaching the input abstract states necessary to expose precision differences or if such inputs are unreachable regardless of sequence length. Lastly, our results indicate that many of these witnesses required multi-instruction sequences to manifest observable differences (column 6), underscoring the importance of bounded program synthesis beyond

Table 4: Precision gains from applying the reduction operator (BPF\_SYNC) to arithmetic eBPF instructions (Linux v6.10). Operators with reduction ( $F_R^\#$ ) are consistently at least as precise as those without ( $F_{-R}^\#$ ), and for some inputs strictly more so (cols. 2–3). Precision testing shows the reduction operator can improve precision in almost all abstract domains (col. 4). Our synthesizer is able to produce witness programs for most of these domains which require four or less instructions to expose precision gaps (cols. 5–6). This demonstrates both the broad utility of reduction and the effectiveness of our approach.

eBPF insn	$F_R^\# \preceq F_{-R}^\#$	$F_{-R}^\# \preceq F_R^\#$	Improved domains?	Synthesized witness for	Witness length?
BPF_RSH	✓	×	<i>All</i>	$A_{u64}, A_{u32},$ $A_{s64}, A_{s32}$	$\leq 4$
BPF_ARSH	✓	×	<i>All</i>	$A_{u64}, A_{u32},$ $A_{s64}, A_{s32}$	$\leq 4$
BPF_LSH	✓	×	<i>All</i>	$A_{u64}, A_{u32},$ $A_{s64}, A_{s32}$	$\leq 4$
BPF_ADD	✓	×	<i>All</i>	<i>All</i>	$\leq 3$
BPF_SUB	✓	×	<i>All</i>	<i>All</i>	$\leq 3$
BPF_AND	✓	×	<i>All</i>	$A_{u64}, A_{s64}$	$\leq 4$
BPF_OR	✓	×	$A_{u64}, A_{u32},$ $A_{s64}, A_{s32}$	$A_{u64}, A_{s64}$	$\leq 4$
BPF_XOR	✓	✓	<i>None</i>	—	—

single-instruction analysis. Taken together, our evaluation demonstrates that the reduction operator is an effective, general-purpose mechanism for improving the precision of value tracking in the eBPF verifier.

## 6 Related Work

**Verifying correctness of the eBPF verifier.** This paper is closest in approach to our prior work verifying the value tracking of the eBPF verifier [65,73]. These automatically verify the soundness of abstract operators by generating verification conditions directly from C code of the Linux eBPF verifier. When a proof of correctness is unattainable, our prototype, Agni [72], generates witness programs illustrating unsound behavior in the verifier. This paper is similar in its approach, but our main focus here is proving precision properties of abstract operator implementations rather than proving their correctness, with the goal of reducing verifier rejections of safe programs. As such, we adapt our differential synthesis to produce witness programs illustrating precision gaps between operators.

**Synthesizing abstract operators.** Manually crafting sound and precise abstract operators is non-trivial and error-prone. Research on developing automated approaches is ongoing [36, 62, 63, 69]. Amurth [43] automatically synthesizes abstract operators for non-relational domains with a user provided DSL. Amurth explores the search space using a dual CEGIS [66] loop with positive (soundness) and negative (precision) counter-examples guaranteeing that when an abstract operator is synthesized, it is sound and the most precise possible in the given language. This approach is extended for reduced-product domains [44]. Our work complements these efforts, allowing precision comparison of operators generated automatically for non-relational domains. Our precision comparison framework along with automated approaches to generating abstract operators could provide a sound and precise foundation for abstract operators in the eBPF verifier.

**Domain refinement and abstract interpretation** Our work builds on foundational ideas in abstract interpretation [30, 32, 33], particularly the use of lattice-theoretic structures to reason about precision. We focus on non-relational domains such as intervals and bitwise abstractions [52, 53], and define precision comparisons over abstract operators using their ordering in the lattice. Our witness generation approach led us to explore how the eBPF verifier combines information from multiple domains, which is non-standard but loosely follows the literature on reduced products domain refinements [33, 53]. Improving cross-domain interaction remains an area of interest [21, 27, 29, 70]. For a broader overview, we refer the reader to an existing survey on product operators [28].

## 7 Conclusion

The eBPF verifier’s *precision* is vital for practical eBPF deployment: overly coarse abstract operators cause the kernel to reject programs that are, in fact, safe. In this work, we propose a systematic approach that leverages formal specifications and program synthesis to compare and validate the precision of competing abstract operator implementations. We introduce a formal precision specification rooted in abstract interpretation, allowing us to precisely define and verify precision relationships between competing operator implementations. When precision gaps are identified, our approach synthesizes concrete eBPF programs demonstrating these differences in real executions. We applied our framework to our improved `bpf_mul` operator and several other precision-related kernel patches, successfully generating witness programs for each, confirming their precision improvements. We propose this methodology as a principled way to ensure that future kernel patches targeting abstract operator precision are effective across all relevant inputs.

## 8 Acknowledgements

This paper is based upon work supported in part by the National Science Foundation under FMITF-Track I Grant No. 2019302 and FMITF-Track II Grant No.

2422076, the Facebook Systems and Networking Award, and the eBPF Foundation Award. We thank the anonymous reviewers for their insightful feedback. We also thank Eduard Zingerman and Alexei Starovoitov for their feedback on our patches.

## References

1. bpf: fix a verifier failure with xor. <https://lore.kernel.org/bpf/20200825064608.2017937-1-yhs@fb.com/>
2. bpf: fix incorrect sign extension in check\_alu\_op(). <https://github.com/torvalds/linux/commit/95a762e2c8c942780948091f8f2a4f32fce1ac6f>
3. bpf: Fix reg\_bound\_offset 64->32 var\_off subreg propagation. <https://lore.kernel.org/linux-patches/20230508094431.898575322@linuxfoundation.org/>
4. bpf, Harden and/or xor value tracking in verifier. <https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf-next.git/commit/?id=1f586614f3ff>
5. bpf selftests. <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/tools/testing/selftests/bpf/>
6. bpf: Verifier, adjust\_scalar\_min\_max\_vals to always call update\_reg\_bounds(). <https://lkml.iu.edu/hypermail/linux/kernel/2208.1/01778.html>
7. bpf, verifier: Improve precision of BPF\_MUL. <https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf-next.git/commit/?id=9aa0ebde0014>
8. bpf, verifier: improve signed ranges reasoning for BPF\_AND. <https://lore.kernel.org/bpf/20240719110059.797546-6-xukuohai@huaweicloud.com/>
9. bpf, x32: Fix bug with ALU64 LSH, RSH, ARSH BPF\_X shift by 0. <https://github.com/torvalds/linux/commit/68a8357ec15bdce55266e9fba8b8b3b8143fa7d2>
10. CVE-2017-16996 Mishandling of register truncation. <https://nvd.nist.gov/vuln/detail/CVE-2017-16996>
11. CVE-2017-17852 Mishandling of 32-bit ALU ops. <https://nvd.nist.gov/vuln/detail/CVE-2017-17852>
12. CVE-2017-17853 Mishandling of 32-bit ALU ops. <https://nvd.nist.gov/vuln/detail/CVE-2017-17853>
13. CVE-2017-17864 Mishandled comparison between pointer and unknown data types. <https://nvd.nist.gov/vuln/detail/CVE-2017-17864>
14. CVE-2018-18445 Mishandling of 32-bit RSH op. <https://nvd.nist.gov/vuln/detail/CVE-2018-18445>
15. CVE-2020-8835 Mishandling of bounds tracking for 32-bit JMPs. <https://nvd.nist.gov/vuln/detail/CVE-2020-8835>
16. CVE-2021-3490 The eBPF ALU32 bounds tracking for bitwise ops (AND, OR and XOR) in the Linux kernel did not properly update 32-bit bounds. CVE-2021-3490
17. Facebook’s Katran load balancer: Kernel XDP program. [https://github.com/facebookincubator/katran/blob/master/katran/lib/bpf/balancer\\_kern.c](https://github.com/facebookincubator/katran/blob/master/katran/lib/bpf/balancer_kern.c)
18. Merge branch 'bpf-register-bounds-logic-and-testing-improvements'. <https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf-next.git/commit/?id=cd9c127069c040d6b022f1ff32fed4b52b9a4017>
19. Netconf 2018 day 1. <https://lwn.net/Articles/757201/>
20. bpf, tnums: Provably sound, faster, and more precise algorithm for tnum\_mul. [Online, Retrieved Oct 19, 2022.] <https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf-next.git/commit/?id=05924717ac70> (2021)

21. Amadini, R., Jordan, A., Gange, G., Gauthier, F., Schachte, P., Søndergaard, H., Stuckey, P.J., Zhang, C.: Combining string abstract domains for javascript analysis: An evaluation. In: Legay, A., Margaria, T. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems*. pp. 41–57. Springer Berlin Heidelberg, Berlin, Heidelberg (2017)
22. Bhat, S., Shacham, H.: Formal verification of the linux kernel ebpf verifier range analysis. <https://sanjit-bhat.github.io/assets/pdf/ebpf-verifier-range-analysis22.pdf> (2022)
23. Borkmann, D.: bpf: Fix scalar32\_min\_max\_or bounds tracking. <https://github.com/torvalds/linux/commit/5b9fbef75b6a98955f628e205ac26689bcb1383e> (2020)
24. Borkmann, D.: bpf: Undo incorrect \_\_reg\_bound\_offset32 handling. <https://git.kernel.org/pub/scm/linux/kernel/git/netdev/net-next.git/commit/?id=f2d67fec0b43edce8c416101cdc52e71145b5fef> (2020)
25. Borkmann, D.: bpf: Fix alu32 const subreg bound tracking on bitwise operations. <https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf.git/commit/?id=049c4e13714ecbca567b4d5f6d563f05d431c80e> (2021)
26. Borkmann, D.: bpf: Fix signed\_sub,add32\_overflows type handling. <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=bc895e8b2a64e502fbbba72748d59618272052a8b> (2021)
27. Cheng, X., Wang, J., Sui, Y.: Precise sparse abstract execution via cross-domain interaction. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. ICSE '24*, Association for Computing Machinery, New York, NY, USA (2024). <https://doi.org/10.1145/3597503.3639220>, <https://doi.org/10.1145/3597503.3639220>
28. Cortesi, A., Costantini, G., Ferrara, P.: A Survey on Product Operators in Abstract Interpretation. *Electronic Proceedings in Theoretical Computer Science* **129**, 325–336 (sep 2013). <https://doi.org/10.4204/eptcs.129.19>
29. Cousot, P., Cousot, R.: Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and per analysis of functional languages). In: *Proceedings of 1994 IEEE International Conference on Computer Languages (ICCL'94)*. pp. 95–112 (1994). <https://doi.org/10.1109/ICCL.1994.288389>
30. Cousot, P.: Lecture 13 notes: Mit 16.399, abstract interpretation. [http://web.mit.edu/afs/athena.mit.edu/course/16/16.399/www/lecture\\_13-abstraction1/Cousot\\_MIT\\_2005\\_Course\\_13\\_4-1.pdf](http://web.mit.edu/afs/athena.mit.edu/course/16/16.399/www/lecture_13-abstraction1/Cousot_MIT_2005_Course_13_4-1.pdf) (2005)
31. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: *Proceedings of the 2nd International Symposium on Programming*, Paris, France. pp. 106–130. Dunod (1976)
32. Cousot, P., Cousot, R.: 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*. p. 238–252. POPL '77, Association for Computing Machinery, New York, NY, USA (1977). <https://doi.org/10.1145/512950.512973>
33. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. p. 269–282. POPL '79, Association for Computing Machinery, New York, NY, USA (1979). <https://doi.org/10.1145/567752.567778>
34. Cree, E.: bpf/verifier: fix bounds calculation on BPF\_RSH. <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=4374f256ce8182019353c0c639bb8d0695b4c941> (2017)

35. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. p. 337–340. TACAS’08/ETAPS’08, Springer-Verlag, Berlin, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-78800-3\\_24](https://doi.org/10.1007/978-3-540-78800-3_24)
36. Elder, M., Lim, J., Sharma, T., Andersen, T., Reps, T.: Abstract domains of affine relations. *ACM Trans. Program. Lang. Syst.* **36**(4) (Oct 2014). <https://doi.org/10.1145/2651361>, <https://doi.org/10.1145/2651361>
37. Fabre, A.: L4drop: Xdp ddos mitigations. <https://blog.cloudflare.com/l4drop-xdp-ebpf-based-ddos-mitigations/>
38. Gershuni, E., Amit, N., Gurfinkel, A., Narodytska, N., Navas, J.A., Rinetzky, N., Ryzhyk, L., Sagiv, M.: Simple and precise static analysis of untrusted linux kernel extensions. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 1069–1084. PLDI 2019, Association for Computing Machinery, New York, NY, USA (2019). <https://doi.org/10.1145/3314221.3314590>
39. Horn, J.: Arbitrary read+write via incorrect range tracking in ebpf. <https://bugs.chromium.org/p/project-zero/issues/detail?id=1454>
40. Horn, J.: bpf: fix 32-bit ALU op verification. <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=468f6eafa6c44cb2c5d8aad35e12f06c240a812a> (2017)
41. Horn, J.: bpf: 32-bit RSH verification must truncate input before the ALU op. <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=b799207e1e1816b09e7a5920fbb2d5fcf6edd681> (2018)
42. Hung, H.W., Amiri Sani, A.: Brf: Fuzzing the ebpf runtime. *Proc. ACM Softw. Eng.* **1**(FSE) (Jul 2024). <https://doi.org/10.1145/3643778>, <https://doi.org/10.1145/3643778>
43. Kalita, P.K., Muduli, S.K., D’Antoni, L., Reps, T., Roy, S.: Synthesizing abstract transformers. *Proc. ACM Program. Lang.* **6**(OOPSLA2) (Oct 2022). <https://doi.org/10.1145/3563334>, <https://doi.org/10.1145/3563334>
44. Kalita, P.K., Reps, T., Roy, S.: Synthesizing abstract transformers for reduced-product domains. In: Giacobazzi, R., Gorla, A. (eds.) *Static Analysis*. pp. 147–172. Springer Nature Switzerland, Cham (2025)
45. Kline, E.: Bpf instruction set architecture (isa). <https://www.kernel.org/doc/html/latest/bpf/standardization/instruction-set.html>
46. Li, Y., Niu, W., Zhu, Y., Gong, J., Li, B., Zhang, X.: Fuzzing logical bugs in ebpf verifier with bound-violation indicator. In: ICC 2023 - IEEE International Conference on Communications. pp. 753–758 (2023). <https://doi.org/10.1109/ICC45041.2023.10278676>
47. Lian, Z., Li, Y., Chen, Z., Shan, S., Han, B., Su, Y.: ebpf-based working set size estimation in memory management. In: 2022 International Conference on Service Science (ICSS). pp. 188–195. IEEE (2022)
48. Linux eBPF maintainers: Bounds syncing for abstract registers. <https://github.com/torvalds/linux/blob/v6.0/kernel/bpf/verifier.c#L1565> (2023)
49. Lu, D., Tang, B., Paper, M., Kogias, M.: Towards functional verification of ebpf programs. In: Proceedings of the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions. p. 37–43. eBPF ’24, Association for Computing Machinery, New York, NY, USA (2024). <https://doi.org/10.1145/3672197.3673435>, <https://doi.org/10.1145/3672197.3673435>

50. Lucas Leong: ZDI-20-1440: An incorrect calculation bug in the linux kernel eBPF verifier. <https://www.zerodayinitiative.com/blog/2021/1/18/zdi-20-1440-an-incorrect-calculation-bug-in-the-linux-kernel-ebpf-verifier>
51. Manfred Paul: CVE-2020-8835: Linux kernel privilege escalation via improper eBPF program verification. <https://www.zerodayinitiative.com/blog/2020/4/8/cve-2020-8835-linux-kernel-privilege-escalation-via-improper-ebpf-program-verification>
52. Miné, A.: Abstract domains for bit-level machine integer and floating-point operations. In: WING'12 - 4th International Workshop on invariant Generation. p. 16. Manchester, United Kingdom (Jun 2012), <https://hal.science/hal-00748094>
53. Miné, A.: Tutorial on static inference of numeric invariants by abstract interpretation. *Foundations and Trends® in Programming Languages* 4(3-4), 120–372 (2017). <https://doi.org/10.1561/25000000034>
54. Mohamed, M.H.N., Wang, X., Ravindran, B.: Understanding the security of linux ebpf subsystem. In: *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems*. p. 87–92. APSys '23, Association for Computing Machinery, New York, NY, USA (2023). <https://doi.org/10.1145/3609510.3609822>, <https://doi.org/10.1145/3609510.3609822>
55. Monniaux, D.: Verification of device drivers and intelligent controllers: a case study. In: *Proceedings of the 7th ACM & IEEE international conference on Embedded software*. pp. 30–36 (2007). <https://doi.org/10.1145/1289927.1289937>
56. Moore, R.E.: *Interval analysis*. Prentice-Hall (1966)
57. Nakryiko, A.: BPF register bounds logic and testing improvements. <https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf-next.git/commit/?id=cd9c127069c0> (2023)
58. Onderka, J., Ratschan, S.: 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*. p. 242–262. Springer-Verlag, Berlin, Heidelberg (2022). [https://doi.org/10.1007/978-3-030-94583-1\\_12](https://doi.org/10.1007/978-3-030-94583-1_12)
59. Palmiotti, V.: Kernel pwning with eBPF: a love story. <https://www.graplsecurity.com/post/kernel-pwning-with-ebpf-a-love-story>
60. Peng, C., Jiang, M., Wu, L., Zhou, Y.: 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*. p. 3928–3942. CCS '24, Association for Computing Machinery, New York, NY, USA (2024). <https://doi.org/10.1145/3658644.3690237>, <https://doi.org/10.1145/3658644.3690237>
61. Regehr, J., Duongsaa, U.: Deriving abstract transfer functions for analyzing embedded software. In: *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems*. p. 34–43. LCTES '06, Association for Computing Machinery, New York, NY, USA (2006). <https://doi.org/10.1145/1134650.1134657>
62. Reps, T., Sagiv, M., Yorsh, G.: Symbolic implementation of the best transformer. In: Steffen, B., Levi, G. (eds.) *Verification, Model Checking, and Abstract Interpretation*. pp. 252–266. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
63. Reps, T., Thakur, A.: Automating abstract interpretation. In: Jobstmann, B., Leino, K.R.M. (eds.) *Verification, Model Checking, and Abstract Interpretation*. pp. 3–40. Springer Berlin Heidelberg, Berlin, Heidelberg (2016)



64. Rick Larabee: eBPF and Analysis of the get-rekt-linux-hardened.c Exploit for CVE-2017-16995. <https://ricklarabee.blogspot.com/2018/07/ebpf-and-analysis-of-get-rekt-linux.html>
65. Shachnai, M., Vishwanathan, H., Narayana, S., Nagarakatte, S.: Fixing latent unsound abstract operators in the ebpf verifier of the linux kernel. In: International Static Analysis Symposium. pp. 386–406. Springer (2024)
66. Solar-Lezama, A.: Program sketching. *International Journal on Software Tools for Technology Transfer* **15**(5), 475–495 (2013)
67. Sun, H., Su, Z.: Validating the eBPF verifier via state embedding. In: 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24). pp. 615–628. USENIX Association, Santa Clara, CA (Jul 2024), <https://www.usenix.org/conference/osdi24/presentation/sun-hao>
68. Sun, H., Xu, Y., Liu, J., Shen, Y., Guan, N., Jiang, Y.: Finding correctness bugs in ebpf verifier with structured and sanitized program. In: Proceedings of the Nineteenth European Conference on Computer Systems. p. 689–703. EuroSys ’24, Association for Computing Machinery, New York, NY, USA (2024). <https://doi.org/10.1145/3627703.3629562>, <https://doi.org/10.1145/3627703.3629562>
69. Thakur, A., Reps, T.: A method for symbolic computation of abstract operations. In: Madhusudan, P., Seshia, S.A. (eds.) *Computer Aided Verification*. pp. 174–192. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
70. Toubhans, A., Chang, B.Y.E., Rival, X.: Reduced product combination of abstract domains for shapes. In: International Workshop on Verification, Model Checking, and Abstract Interpretation. pp. 375–395. Springer (2013)
71. Vishwanathan, H., Shachnai, M., Narayana, S., Nagarakatte, S.: Sound, precise, and fast abstract interpretation with tristate numbers. In: Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization. p. 254–265. CGO ’22, IEEE Press (2022). <https://doi.org/10.1109/CGO53902.2022.9741267>
72. Vishwanathan, H., Shachnai, M., Narayana, S., Nagarakatte, S.: Agni: Verifying the Verifier (eBPF Range Analysis Verification). <https://github.com/bpfverif/ebpf-range-analysis-verification-cav23> (2023)
73. Vishwanathan, H., Shachnai, M., Narayana, S., Nagarakatte, S.: Verifying the verifier: ebpf range analysis verification. In: Computer Aided Verification: 35th International Conference, CAV 2023, Paris, France, July 17–22, 2023, Proceedings, Part III. p. 226–251. Springer-Verlag, Berlin, Heidelberg (2023). [https://doi.org/10.1007/978-3-031-37709-9\\_12](https://doi.org/10.1007/978-3-031-37709-9_12)
74. Yang, J., Chen, L., Bai, J.: Redis automatic performance tuning based on ebpf. In: 2022 14th International Conference on Measuring Technology and Mechatronics Automation (ICMTMA). pp. 671–676. IEEE (2022)