Automatic Synthesis of Abstract Operators for eBPF

Harishankar Vishwanathan
harishankar.vishwanathan@rutgers.edu
Rutgers University
Piscataway, USA

Matan Shachnai
m.shachnai@rutgers.edu
Rutgers University
Piscataway, USA

Srinivas Narayana
srinivas.narayana@rutgers.edu
Rutgers University
Piscataway, USA

Santosh Nagarakatte
santosh.nagarakatte@cs.rutgers.edu
Rutgers University
Piscataway, USA

Abstract

This paper proposes an approach to automatically synthesize sound and precise abstract operators for the static analyzer in the eBPF verifier. The eBPF verifier ensures that only safe user-defined programs are loaded into the kernel. An unsound operator can lead to unsafe programs being accepted, while an imprecise operator can cause safe programs to be rejected. Our approach starts by generating candidate operators using input-output examples tailored for the eBPF verifier's abstract operators and iteratively refines it for soundness and precision. Using this approach, we have generated more precise variants of existing operators. Our approach also generates numerous sound and unsound operators that can serve as test suites for existing eBPF verification and fuzzing frameworks.

CCS Concepts

• **Software and its engineering** → **Operating systems**; **Formal software verification**; • **Theory of computation** → *Automated reasoning*; Abstraction.

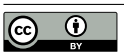
Keywords

eBPF, abstract interpretation, program synthesis

ACM Reference Format:
Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2025. Automatic Synthesis of Abstract Operators for eBPF. In *3rd Workshop on eBPF and Kernel Extensions (eBPF '25), September 8–11, 2025, Coimbra, Portugal.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3748355.3748361

1 Introduction

The eBPF ecosystem has emerged as a general-purpose platform for safely extending the functionality of operating system kernels with user-supplied programs. The safety of this ecosystem hinges on the correctness of the eBPF verifier: a static analyzer that enforces memory safety, type correctness, and bounded control-flow. The verifier symbolically explores every code path in the eBPF program and accepts a program only if it can prove that no path leads to unsafe behavior under any input.



eBPF '25, Coimbra, Portugal

ACM ISBN 979-8-4007-2084-0/2025/09
https://doi.org/10.1145/3748355.3748361

The verifier performs *abstract interpretation* to track the possible values that program variables (*i.e.,* registers in eBPF bytecode) can take across all executions. The verifier uses succinct representations to track register values, termed *abstract domains* [37]. For example, it uses an interval abstract domain to track the minimum and maximum values a register may hold (*i.e.,* `[min, max]`), and a bitwise abstract domain, called tristate numbers, which tracks each individual bit of a program register. Together, the representations of a register from all the abstract domains form the register's *abstract value* at verification time.

The verifier uses *abstract operators* to evolve the abstract state of each register depending on the concrete instructions in the input eBPF program. Consider an instruction in eBPF bytecode such as `r2 = *(u32 *)(r10 - r1)`, which loads a 32-bit word from the stack by subtracting `r1` from the frame pointer `r10`. If the verifier determines that `r1` always lies in the interval `[-10,10]`, it rejects the program because the offset can be potentially negative, resulting in an out-of-bounds stack access. However, if the instruction is preceded by `r1 &= 0xf`, the verifier can update the interval to `[0,15]`, allowing the verifier to prove that the access is safe. Indeed, such code patterns are common in hand-written eBPF code to "guide" the verifier into accepting programs. It is the job of the verifier's abstract operator to compute these updated values soundly and precisely. Soundness refers to the property that a register's abstract value includes all concrete values possible at run time for that register. Precision refers to the property that the abstract value excludes concrete register values that will never occur at run time.

Unsoundness in the verifier has been exploited to perform arbitrary kernel reads/writes and privilege escalation attacks [14, 18, 24, 25]. Unsurprisingly, several academic and industry efforts have focused on testing or formally verifying the soundness of eBPF abstract operators [4, 10, 12, 20, 29, 33, 34, 36, 37]. Apart from soundness, imprecision in the static analyzer can cause valid, safe programs to be rejected, impairing the usability and adoption of eBPF for legitimate applications [5, 8, 23, 39]. For instance, after `r1 &= 0xf`, a sound but imprecise abstract operator might conclude that "bitwise AND with a positive constant always yields a non-negative result" and update the interval for `r1` to `[0,INT_MAX]`. While the runtime value of `r0` will indeed be within this range, this bound is too imprecise: it is not helpful to prove that the memory access will stay within the 512-byte eBPF stack. As a result, the verifier will conservatively reject the program.

Manually crafting sound and precise abstract operators is hard. The numerous patches to improve both the soundness and precision

of eBPF abstractions highlight the importance of this problem [1–3, 5, 8, 13–15, 18, 23–25, 28, 32, 39]. This paper aims to develop techniques to automatically synthesize sound and precise eBPF abstract operators. Our work is inspired by counterexample-guided inductive synthesis (CEGIS) [31], and recent work on synthesizing abstract operators, namely Amurth [16]. CEGIS is a classic framework used to synthesize programs that satisfy a given specification. Amurth builds on CEGIS by introducing separate specifications for soundness and precision to synthesize abstract operators [16, 17].

We design techniques to synthesize sound and optimally precise abstract operators for the abstract domains used in the eBPF verifier. Our approach works by iteratively guessing an abstract operator based on examples of inputs and outputs and then checking that the guessed program satisfies the specifications of soundness and precision (§3). If the operator fails the specification, the underlying solver can produce a counterexample, which is added to the set of examples and used to guess the next abstract operator. The CEGIS iterations guess several abstract operators (which may be unsound or imprecise) along the way, provably converging to a sound and precise abstract operator.

As an added benefit from this iterative procedure, we observe that the intermediate abstract operators produced by the algorithm can be used to craft test cases for existing eBPF verifier checking tools like Agni [37] or Buzzer [10]. We believe that such test cases, many of which include "nearly sound" (but actually unsound) abstract operators, constitute interesting adversarial tests for such tools, helping improve the assurances provided by the checkers.

In summary, our contributions are as follows. (1) Our approach automatically synthesizes sound and precise abstract operators for unsigned intervals and tristate numbers. We are the first to synthesize operators for tristate numbers (§3). (2) We discover new, more precise abstract operators for unsigned interval addition and subtraction (§4). A patch implementing these precision improvements has been merged into the latest Linux kernels [35].

2 Background

Static analysis in the eBPF verifier. To statically prove safety, the eBPF verifier uses abstract interpretation [6], a technique that approximates the set of possible program behaviors for all inputs. At each program point, the verifier maintains a summary of the program state, mapping each program register to its corresponding abstract value. The abstract value of a register conservatively captures some property of that register at the given program point, such as the possible values it may hold, its liveness information, or the kind of memory it may point to. Each abstract value is drawn from an abstract domain, which defines the kinds of properties that can be represented and how they evolve under program operations.

Value tracking and interval domains. The verifier's value tracking analysis aims to conservatively estimate the set of values a program register may hold across all executions. This estimation is typically done using *numerical* abstract domains, such as the interval domain. In this domain, a program register `p` is represented by an abstract interval value $P = [P_l, P_u]$, indicating that $P_l \leq \texttt{p} \leq P_u$ at that program point. The verifier processes each instruction in sequence, updating the interval accordingly. For example, an instruction like `p += 10` would update the interval to $[P_l + 10, P_u + 10]$.

Bitwise reasoning with the tristate domain. To improve precision for bitwise operations, the verifier also employs the *tristate number* (tnum) domain [36], which tracks for each bit, whether it is definitely 0, definitely 1, or unknown (`x`) across all executions. For example, if `p` is a program input, it is completely unknown and therefore represented as `xxx...x` (all `x`s). The instruction `q = p & 0xf` masks out all but the lowest four bits of `p`, yielding `0...0xxxx` for `q`. In the verifier, a tnum is represented using a pair of `u64` integers: `{value, mask}`. A bit is known to be 0 or 1 if the corresponding `mask` bit is 0, in which case the `value` bit gives its concrete value. If the `mask` bit is 1, the bit is unknown (`x`), regardless of the `value` bit. For instance, the tnum from our example `0...0xxxx` is represented with `{value=0, mask=0b1111}`, while the tnum `0...01x` is represented with `{value=0b010, mask=0b001}`.

The tristate domain is particularly effective for reasoning about bitwise operations, where interval domains often lose precision. Together, the interval and tristate domains enable the verifier to prove safety properties.

Concretization and abstraction. The concretization function γ and the abstraction function α relate the concrete behavior of a program to its abstract representation and are used to reason about the correctness of approximation. Function γ maps an abstract value to the set of concrete values it represents (*e.g.,* $\gamma([3,5]) = \{3,4,5\}$), while α maps a set of concrete values to the most precise abstract value that overapproximates them (*e.g.,* $\alpha(\{3,4,7\}) = [3,7]$).

Abstract operators. An abstract operator F takes abstract inputs and approximates the effect of applying the corresponding concrete operator f to all possible concrete values those inputs represent. An abstract operator must be sound and precise. Soundness requires that, for any input(s) to F, the output must include all possible concrete results of applying f to values drawn from the input(s). Precision requires that the output must exclude concrete outputs that are not a result of applying f to values drawn from the input(s). The ideal operator computes $\alpha \circ f \circ \gamma$, *i.e.,* it concretizes the inputs using γ, applies f, and then abstracts the result using α. This computation yields the most precise and sound result in the given abstract domain. Any abstract operator that computes $\alpha \circ f \circ \gamma$ *exactly* is called the optimally precise operator for f in that domain.

However, computing $\alpha \circ f \circ \gamma$ exactly is often impractical. For instance, performing the addition operation on two intervals $P = [50, 100]$ and $Q = [150, 200]$ requires evaluating 2601 combinations of $x + y$ where $x \in \gamma([50, 100])$ and $y \in \gamma([150, 200])$, and then abstracting the result using α. This exhaustive enumeration is clearly inefficient. Instead, most domains define efficient closed-form functions, which may overapproximate $\alpha \circ f \circ \gamma$. In the interval domain, for instance, the addition of two intervals $[P_l, P_u]$ and $[Q_l, Q_u]$ is implemented as $[P_l + Q_l, P_u + Q_u]$. This expression is, in fact, optimal when reasoning over unbounded integers because it computes $\alpha \circ f \circ \gamma$ exactly. However, in the case of bounded machine integers, *e.g.,* 8-bit unsigned integers, this computation works only when no overflow occurs. In cases like $[50, 100] + [150, 200]$, some concrete sums exceed the unsigned 8-bit maximum of 255. The abstract operator must conservatively return $[0, 255]$ to remain sound, sacrificing precision.

Automated program synthesis. Program synthesis is the task of automatically generating programs that satisfy a user-provided specification of their input-output behavior. Specifications can take the form of general correctness properties or input-output examples, and are typically encoded as logical constraints that the synthesized program must satisfy. Formally, given a specification ϕ_{spec}, we seek a program $\mathcal{P}$ such that, for all inputs $\mathcal{I}$ and outputs $\mathcal{O} = \mathcal{P}(\mathcal{I})$, $\phi_{\text{spec}}(\mathcal{I}, \mathcal{O})$ holds. This process is challenging because synthesis must search over the space of all possible programs, which grows exponentially with the length of programs considered.

Counterexample-guided inductive synthesis (CEGIS). To make the search tractable, a common strategy is to use counterexamples provided by an SMT solver to iteratively refine candidate programs that fail to meet the specification. CEGIS [31] is an iterative technique that breaks the synthesis task into two alternating phases: (1) synthesize a candidate program that works on a *finite set* of inputs, and (2) verify whether the candidate satisfies the specification *for all* inputs. The process begins with a finite set of inputs $\mathcal{S}$ and uses an SMT solver to synthesize a candidate program $\mathcal{P}_c$ that satisfies the specification ϕ_{spec} for the inputs in $\mathcal{S}$. It then verifies if this $\mathcal{P}_c$ generalizes correctly (*i.e.,* it correctly satisfies ϕ_{spec} for all inputs), also using an SMT solver. If the verification fails for $\mathcal{P}_c$, the solver returns a counterexample, *i.e.,* an input for which the program is incorrect. This counterexample is added to $\mathcal{S}$, and the loop repeats. Over time, the example set grows, guiding synthesis towards programs that generalize correctly to more inputs. The process terminates when the candidate $\mathcal{P}_c$ verifies successfully, *i.e.,* it satisfies the specification ϕ_{spec} for all inputs.

Component-based synthesis. To synthesize programs, we must define a language in which programs can be expressed. One approach is to use a library of *components*, which are low-level building blocks such as arithmetic and bitwise functions [11]. Each component specifies its arity and input-output semantics, expressed as logical constraints. Programs are constructed by composing components. The synthesis task becomes assigning each component to a specific line number and finding valid placements and dataflow between the inputs and outputs of the components. These requirements are encoded as constraints over each component's line number assignments and solved with the help of an SMT solver, typically using CEGIS.

3 Abstract Operator Synthesis

We aim to automatically synthesize sound and precise abstract operators for use in the eBPF verifier. Given a concrete eBPF operator f, we seek an abstract operator F that conservatively overapproximates f (*i.e.,* soundness) while excluding as many unreachable results as possible (*i.e.,* precision). For illustration, we assume f and F are binary operators (*i.e.,* take two inputs and return one output). Our algorithm, presented in Figure 1, is inspired by Amurth [16]. It contains two CEGIS loops and proceeds in two steps. Step A uses a CEGIS procedure with a soundness specification to synthesize a sound candidate operator. Step B then iteratively refines this candidate also using a CEGIS procedure, guided by a precision specification, to produce a more precise operator that remains sound.

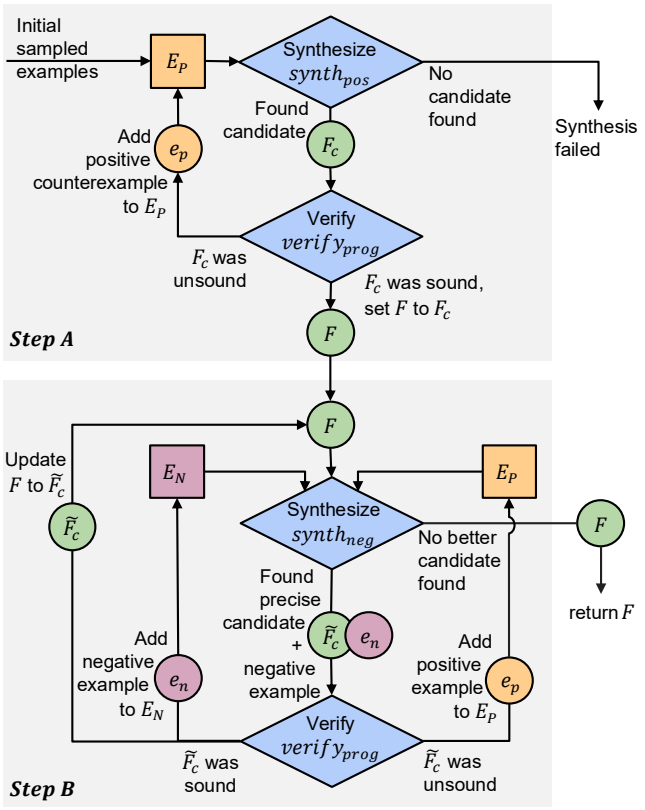


Figure 1: **Diagram illustrating our algorithm for synthesizing an abstract operator. Step A synthesizes candidates F_c guided by (positive) counterexamples. When a sound candidate is found, it is set to F and passed to Step B. Step B refines F to synthesize more precise candidates $\widetilde{F_c}$, along with negative examples e_n that $\widetilde{F_c}$ excludes, but F includes. If $\widetilde{F_c}$ is sound, F is updated to $\widetilde{F_c}$, and the loop continues. When no further refinement is possible, the algorithm returns F.**

3.1 Overview of the Algorithm

Step A: Candidate search based on soundness. This step aims to synthesize a sound candidate operator F corresponding to f. It can be seen as an application of classic CEGIS to the setting of abstract operator synthesis using a specification of abstract operator soundness (§2). We express the soundness requirement using *positive examples*: tuples (P, Q, z) where P and Q are abstract inputs, such that z must be contained in $F(P, Q)$, and z is a concrete value obtained as a result of applying f to concrete values drawn from P and Q. Positive examples constrain the input-output behavior of F by requiring that its output contains the specified concrete value. We initialize the CEGIS loop with randomly generated positive examples. The synthesis phase encodes the soundness constraints and queries an SMT solver for a candidate F_c, constructed from our fixed library of components. Then, the verification phase checks whether there exists any new tuple (P_1, Q_1, z_1) such that $z_1 \notin F_c(P_1, Q_1)$, where z_1 is a concrete value obtained as a result of applying f to concrete values drawn from P_1 and Q_1. If so, this counterexample is added to the example set, and the loop repeats. The loop terminates when F_c is sound. By the end of Step A, F is established as F_c, our first sound abstract operator for f.

Step B: Refinement of candidates based on precision. Soundness alone is insufficient. A candidate operator F for the unsigned interval domain obtained from Step A, that always returns the interval `[0, UINT_MAX]` regardless of the input, is trivially sound because it always includes all possible concrete results. Instead, we want the abstract operator to exclude concrete results that cannot occur, thereby producing tighter output intervals. To improve precision, we borrow the idea of negative examples from Amurth [16], by writing down a precision specification that identifies inputs where the current operator F is overly conservative. Specifically, we ask: is there a new tuple (P_2, Q_2, z_2) where $z_2 \in F(P_2, Q_2)$, such that a more precise operator $\widetilde{F_c}$ excludes z_2, *i.e.,* $z_2 \notin \widetilde{F_c}(P_2, Q_2)$, while including all the positive examples in E_p and remaining sound. Such an example is called a negative example because it highlights a spurious result that the operator should avoid.

Step B also uses a CEGIS loop. The synthesis phase searches for a more precise operator $\widetilde{F_c}$ and a negative example (P, Q, z) that it excludes. However, $\widetilde{F_c}$ may be unsound. So we verify $\widetilde{F_c}$, as in Step A. If a soundness violation is found, we add the corresponding counterexample to the set of positive examples. If no such violation exists, we promote $\widetilde{F_c}$ as the new candidate, save the negative example, and continue refining. This loop continues until no further improvements are possible or a specified time budget is exhausted. By the end of Step B, F has been refined to a sound and precise abstract operator for f.

3.2 Illustration of the Algorithm

We illustrate the algorithm by synthesizing an abstract operator F corresponding to a concrete operator f for 8-bit unsigned integer addition ($+_{u8}$, following standard C semantics). We assume a component-based synthesis setting and a library of $\{+, -, \times, k\}$ components, where k is a constant integer literal. To bootstrap the synthesis, we populate a set E_p with positive examples (P_i, Q_i, z_o) by generating random abstract values P_i and Q_i, sampling concrete values $x \in P_i$ and $y \in Q_i$, and finally setting $z_o = f(x, y)$. For example, the tuple $([2, 5], [4, 8], 10)$ is a positive example for unsigned interval addition because $4 \in [2, 5]$, $6 \in [4, 8]$, and $4 + 6 = 10$.

Synthesizing a candidate operator using $synth_{pos}$. We want to synthesize a candidate that satisfies the examples in E_p using our library of components. The SMT formula for synthesizing such a candidate F_c is as follows (Eqn. 1).

$$synth_{pos}(E_p) \triangleq \exists F_c : \bigwedge_{(P,Q,z) \in E_p} z \in F_c(P, Q) \tag{1}$$

An example of this query, based on some positive examples, is: $\exists F_c : 10 \in F_c([2, 5], [4, 8]) \wedge 5 \in F_c([1, 1], [4, 4])$. Suppose the solver comes up with a candidate $F_c(P, Q) = [0, 2 \times Q_u]$ because $10 \in [0, 16]$, and $5 \in [0, 8]$.

Verifying the soundness of F_c using $verify_{prog}$. Our candidate F_c may be unsound. We check the soundness of F_c by constructing a formula that asks if there exist any input abstract values P and Q and concrete values $x \in P$ and $y \in Q$, such that F_c's output does not contain $f(x, y)$, as shown in Eqn. 2.

$$verify_{prog}(F_c) \triangleq \exists x, y, P, Q : x \in P \wedge y \in Q \wedge f(x, y) \notin F_c(P, Q) \tag{2}$$

If no such values x, y, P, Q exist, our candidate operator F_c is sound for all inputs. However, if such values do exist, $e_p = (P, Q, f(x, y))$ constitutes a positive counterexample for the unsoundness of F_c because F_c's output does not include e_p. For example, if $F_c(P, Q) = [0, 2 \times Q_u]$, the solver might come up with $([5, 5], [2, 2], 7)$ as a counterexample. We collect such positive examples in E_p.

Arriving at a sound candidate F. Step A of our algorithm employs a CEGIS loop by calling $synth_{pos}$ and $verify_{prog}$ in succession. Whenever $synth_{pos}$ gives us a candidate operator F_c, we proceed to check its soundness using $verify_{prog}$. If $verify_{prog}$ gives us a counterexample, we add it to E_p and try to synthesize a new candidate. When $verify_{prog}$ does not return a counterexample, we have a sound candidate operator. The sound candidate F_c obtained at the end of Step A is set to F and passed to Step B.

Synthesizing a more precise operator $\widetilde{F_c}$. We aim for our operator F to approximate $\alpha \circ f \circ \gamma$ as closely as possible (§2). Suppose we obtained a candidate operator $F(P, Q) = [0, P_u + Q_u]$ from the previous Step A. While F is sound, it significantly overapproximates the output. We use negative examples to improve the precision of F.

A negative example is only meaningful in the context of a pair of abstract operators. The example serves as a witness to the precision improvement of one operator over the other. Consider example $e_n : ([2, 4], [3, 5], 1)$. Operator $F(P, Q) = [0, P_u + Q_u]$ includes e_n since $1 \in [0, 9]$, whereas operator $\widetilde{F}(P, Q) = [P_l + Q_l, 255]$ excludes e_n because $1 \notin [5, 255]$. Thus, e_n is a negative example witnessing the improved precision of $\widetilde{F}$ over F.

Our task is to synthesize a more precise operator than the sound operator F we obtained from Step A. In Step A, we iteratively collected positive examples in E_p. Since F is sound, we know that it includes all the positive examples in E_p. Hence, a more precise operator $\widetilde{F_c}$ must include all the positive examples in E_p, and additionally exclude a new negative example e_n that F includes. We use the following query (Eqn. 3) to find an operator $\widetilde{F_c}$ and the corresponding negative example $e_n = (P_i, Q_i, z_o)$, such that the current best operator we have (F) includes the concrete value z_o in its output when given inputs P_i and Q_i, but $\widetilde{F_c}$'s output excludes it.

$$\exists \widetilde{F_c}, P_i, Q_i, z_o : z_o \notin \widetilde{F_c}(P_i, Q_i) \wedge z_o \in F(P_i, Q_i) \wedge \bigwedge_{(P,Q,z) \in E_p} z \in \widetilde{F_c}(P, Q) \tag{3}$$

Verifying soundness of $\widetilde{F_c}$. Our new candidate operator $\widetilde{F_c}$ need not be sound. We call $verify_{prog}$ to check its soundness. If $verify_{prog}$ returns a counterexample e_p, $\widetilde{F_c}$ is unsound. We add e_p to the set E_p, ensuring we do not discover the same $\widetilde{F_c}$ again. For example, let us say that after Step A, we had our sound candidate operator $F(P, Q) = [0, P_u + Q_u]$, and $E_p = \{([2, 5], [4, 8], 10), ([1, 1], [4, 4], 5)\}$. In Step B, we might synthesize an operator $\widetilde{F_c}(P, Q) = [0, 2 + Q_u]$ and $e_n = ([4, 6], [3, 5], 9)$ because 9 is excluded by $\widetilde{F_c}([4, 6], [3, 5])$, but included by $F([4, 6], [3, 5])$. When we call $verify_{prog}(\widetilde{F_c})$, we find that $\widetilde{F_n}$ is unsound, and we may obtain a counterexample such as $([5, 5], [6, 6], 11)$, which is then added to E_p.

When $verify_{prog}$ does not return a counterexample, $\widetilde{F_c}$ is sound. We add the negative example e_n to E_n, update our candidate F to $\widetilde{F_c}$, and continue the loop to explore an even more precise operator.

Using $synth_{neg}$ to synthesize new operators. During synthesis, the goal is to ensure that each new candidate we synthesize is more precise than the current candidate. The current candidate F already excludes the last discovered negative example. To improve upon its precision, a new candidate $\widetilde{F_c}$ must continue to exclude all previously discovered negative examples in E_n, include all the positive examples in E_p, and further exclude one additional negative example relative to F. Hence, we use $synth_{neg}$ in our algorithm (Eqn. 4), which encodes this constraint. At the beginning of Step B, when there are no negative examples in E_n, $synth_{neg}$ simplifies to Eqn. 3.

$$synth_{neg}(F, E_p, E_n) \triangleq$$
$$\exists \widetilde{F_c}, P_i, Q_i, z_o : z_o \notin \widetilde{F_c}(P_i, Q_i) \wedge z_o \in F(P_i, Q_i) \wedge \bigwedge_{(P,Q,z)\in E_p} z \in \widetilde{F_c}(P,Q) \bigwedge_{(P,Q,z)\in E_n} z \notin \widetilde{F_c}(P,Q) \quad (4)$$

Terminating with a precise operator. Step B uses the CEGIS paradigm to iteratively construct more precise operators that are sound using $synth_{neg}$ and $verify_{prog}$. The algorithm terminates when $synth_{neg}$ fails to find a more precise operator than the current candidate F. At that point, the current candidate F is returned.

Discovering optimally precise operators. If the maximally precise operator is expressible in our library of components, then synthesizing negative examples from any candidate F that overapproximates it will yield a new abstract operator and a new negative example that witnesses the overapproximation, enabling further refinement. Hence, our algorithm will eventually converge to the optimally precise operator (once no such refinements are possible). However, when the optimally precise operator is not expressible, there may exist multiple sound abstract operators that are incomparable in precision. For example, suppose the library only permits operators that return $[0, a{+}b]$ or $[a{+}b, 255]$, where a, b are interval endpoints. In that case, $F_1(P,Q) = [0, P_u + Q_u]$ and $F_2(P,Q) = [P_l + Q_l, 255]$ are both sound but incomparable: each is more precise than the other on a different subset of inputs. Our algorithm may settle on a locally optimal operator, *i.e.,* one that is sound and cannot be refined further without violating soundness. This is the best any algorithm can do with the provided library of components.

Intermediate operators. Our algorithm discovers unsound operators that nonetheless satisfy many positive and negative examples. These intermediate operators are valuable test inputs for verifier testing frameworks, as they are nearly sound over a large portion of the input space. Further, rerunning synthesis with a different random seed generates a different set of positive examples E_p, biasing the search toward different regions of the program space, yielding sound yet incomparably precise and diverse unsound operators.

3.3 Scaling Synthesis

Component-based synthesis scales poorly: the synthesis constraints grow quadratically with the number of components [9, 11], increasing the time to solve them. To improve scalability, we leverage knowledge about the abstract domain and the concrete operator to introduce library components that encapsulate the actions of multiple primitive components. We call such components *CISC components,* akin to complex instructions in a hardware architecture.

Abstract operator	Components (#)		CISC components	Time (m)	Ops (#)	
	Base	CISC			S	U
`tnum_or`	12	–	–	< 1	4	2
`tnum_xor`	12	–	–	< 1	3	2
`tnum_and`	12	–	–	41	4	1
`tnum_add`	14	10	or_3, and_{neg}	231	4	4
`tnum_sub`	14	10	or_3, and_{neg}	886	5	5
`uint_add`	15	13	add^1_{of}, add^2_{of}, sel_{add}	30	10	10
`uint_sub`	15	13	sel_{sub}	106	8	8

$and_{neg}(a,b) \triangleq a \,\&\, {\sim}b$ $\quad or_3(a,b,c) \triangleq a \mid b \mid c$
$add^1_{of}(a,b) \triangleq (a \le a+b)\ ?\ 1 : 0$ $\quad add^2_{of}(a,b) \triangleq (a > a+b)\ ?\ 1 : 0$
$sel_{add}(a,b,c,d) \triangleq a\ ?\ b : (c+d)$ $\quad sel_{sub}(a,b,c,d) \triangleq a\ ?\ b : (c-d)$

Table 1: **Synthesis results for tristate (`tnum`) and unsigned interval (`uint`) abstract operators. Columns (L—R): operator name; number of components for Base and CISC configurations; CISC components used (if any); synthesis time (in minutes); and number of sound (S) and unsound (U) operators synthesized.**

For example, we know an abstract addition operator will need to detect overflow, which for two machine integers x and y can be expressed as $x > x + y$. Instead of using a basic if-then-else conditional component like $ite(o,a,b) \triangleq o\ ?\ a : b$ (where o is a single boolean input), our CISC version supports compound conditions directly: $ite'(o_1, o_2, a, b) \triangleq (o_1 > o_1 + o_2)\ ?\ a : b$. This approach helps reduce the number of components in the library and the length of candidate programs, enabling us to synthesize candidate operators for eBPF domains like unsigned intervals and tnums.

4 Evaluation

Our prototype is tailored to synthesize abstract operators for the eBPF verifier. It implements the algorithm in Figure 1 using component-based synthesis and the Z3 SMT solver [7]. It is open source and publicly available [38].

We evaluate the effectiveness of our algorithm by synthesizing abstract operators for the tristate number (`tnum`) and unsigned interval (`uint`) domains. Each synthesis instance is provided with a component library, a concrete operator f, and a concretization function $\gamma_{\mathbb{A}}$ for abstract domain $\mathbb{A}$. We use f and $\gamma_{\mathbb{A}}$ to construct the formulae described in §3, which are then passed to the SMT solver. All calls to the SMT solver were given a timeout of 24 hours. We want to evaluate our approach on the following: (1) the effectiveness in synthesizing operators for the `tnum` and `uint` domains, (2) the ability to discover precise operators, and (3) the ability to produce interesting operators for testing verifier testing frameworks.

Ability to synthesize abstract operators. Table 1 presents the list of operators that were synthesized with our approach. Abstract operators for the tristate and interval domains require a small number of components (at most 15). Our prototype is able to synthesize abstract operators for bitwise `or`, bitwise `xor`, and bitwise `and` in the tristate domain within 1 hour. In contrast, our prototype could not synthesize an operator for `add` and `sub` even after 24 hours because they need relatively more components. After introducing CISC components (§3.3), our prototype is able to synthesize the abstract operators for `add` and `sub` in the tristate domain and also the ones for `add` and `sub` in the unsigned interval domain.

```
loverflows = Pl > Pl + Ql
uoverflows = Pu > Pu + Qu
if loverflows or uoverflows:
    return [0, UMAX]
else:
    return [Pl + Ql, Pu + Qu]
```

```
uoverflows = Pu > Pu + Qu
loverflows = Pl > Pl + Ql
if uoverflows and not loverflows:
    return [0, UMAX]
else:
    return [Pl + Ql, Pu + Qu]
```

Figure 2: **Addition operators in the unsigned interval domain. L: The eBPF implementation. R: More precise implementation synthesized by our algorithm. The inputs to the operators are abstract interval values** a = [Pl, Pu] **and** b = [Ql, Qu]**.**

Ability to generate precise operators. For add and sub in the unsigned interval domain, our prototype synthesized a more precise abstract operator than existing implementations in the eBPF verifier [21, 22]. Consider two input intervals P = [Pl, Pu] and Q = [Ql, Qu] in this domain. Figure 2-L shows the existing eBPF operator for add, and Figure 2-R shows our new operator, which leverages the wrap-around semantics of unsigned overflow. Specifically, there are three possible cases for adding any $x \in P$ and $y \in Q$. (1) No Overflow: even the largest sum Pu + Qu ≤ UINT_MAX. No $x + y$ addition overflows. (2) Partial Overflow: the smallest sum Pl + Ql ≤ UINT_MAX is within range, but the largest sum Pu + Qu > UINT_MAX. Some additions $x + y$ overflow, and some do not. (3) Full Overflow: even the smallest sum Pl + Ql > UINT_MAX. Every addition $x + y$ overflows, and all the results wrap around.

While the existing eBPF operator returns the unbounded interval [0, UINT_MAX] whenever any overflow is possible, *i.e.,* cases (2) and (3), our synthesized operator returns [0, UINT_MAX] only in case (2). It safely returns the interval [Pl + Ql, Ql + Qu] in case (1) (since no overflow occurs) and case (3) (since all results overflow and the returned interval captures the wrapped-around additions). This strategy yields strictly more precise results than the baseline operator. For example, considering 8-bit integers, given intervals P = [163, 215] and Q = [156, 213], all concrete sums overflow. The baseline operator returns [0, 255], whereas our synthesized operator produces the more precise [63, 172].

Our prototype also synthesized an operator for sub in the interval domain that is more precise than the existing implementation. The reasoning for the precision improvement is similar, but applied to underflow instead of overflow. Our patch that implements the operator from Figure 2-R, as well as the operator for interval subtraction, has been merged into the latest Linux kernels [35].

Generating tests for various frameworks. Our prototype generates a rich variety of abstract operators, which can be used to test the eBPF ecosystem. The last column of Table 1 shows the number of sound and unsound operators synthesized by our prototype in one iteration. For example, while synthesizing the operator for bitwise and in the tristate domain, our algorithm discovered one unsound and four sound intermediate operators. Table 2 shows three of the sound synthesized operators for and in the tristate domain, each more precise than the ones to its left. Rerunning our prototype with different random seeds yields an even broader set of operators. For example, over 10 runs with different seeds, our prototype cumulatively discovered 49/12 (sound/unsound) tnum_and, 36/11 tnum_or, and 27/16 tnum_xor operators.

Testing sound and unsound operators using Agni. We substituted the sound and unsound operators generated by our prototype into the Linux eBPF verifier's C code and tested these operators using Agni [37], a framework for testing the soundness of operators. We tested two sound and two unsound operators for each of tnum and, or, xor, add, and sub. Agni verified the correctness of the sound operators within a minute and flagged the unsound ones within 10 minutes. In general, the sound and unsound operators synthesized by our approach can serve as targeted test cases for evaluating and improving eBPF verifier testing tools [10, 12, 33, 34].

o1 = bv & bm	o1 = am & av	o1 = bv & av
o2 = bv | av	o2 = am | av	o2 = am | av
o3 = o2 | am	o3 = bm | bv	o3 = am & bv
return (o1, o3)	o4 = o2 & o3	o4 = o3 | bm
	return (o1, o4)	o5 = o2 & o4
		return (o1, o5)

Table 2: **Three sound candidate abstract operators for tristate bitwise and, in increasing order of precision (left to right). The inputs are tristate abstract values** a = (av, am) **and** b = (bv, bm)**.**

5 Related Work

To our knowledge, this paper represents the first effort that systematically attempts to improve the precision of the Linux kernel's eBPF abstract operators. Our approach is closely related to Amurth [16], which also synthesizes abstract operators. It generalizes the prior work on synthesizing abstract operators for low-level machine code verification [26, 27]. We borrow the core idea of refining precision using negative examples within a CEGIS loop from Amurth. Amurth uses two non-deterministically interleaved CEGIS loops for soundness and precision. Negative examples are added optimistically to improve precision, even if they conflict with prior positive examples. This design allows Amurth to aggressively pursue precision improvements. To resolve resulting inconsistencies, Amurth invokes a MaxSAT solver [19] to retain a maximal subset of compatible negative examples. In contrast, our algorithm adds a negative example only if it preserves soundness, ensuring consistency of examples by design and avoiding the need for MaxSAT. This yields a more straightforward, deterministic refinement process. In addition, Amurth uses the Sketch program synthesis framework [30], while we adopt a component-based synthesis approach [11].

6 Conclusion

We present an approach to automatically synthesize sound and precise abstract operators for use in the eBPF verifier. The synthesized operators match or improve the precision of existing implementations. Further, numerous sound and unsound operators synthesized during this process can serve as test cases for the ecosystem of verifiers and fuzzers for the eBPF verifier. Our results enhance both the verifier's core analyses and the infrastructure used to test and validate them.

7 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 valuable feedback. We also thank Eduard Zingerman for his feedback on our patches. This work does not raise any ethical concerns.

References

[1] 2017. CVE-2017-16996: Mishandling of register truncation. https://nvd.nist.gov/vuln/detail/CVE-2017-16996. (2017). Accessed: 2025-05-05.

[2] 2017. CVE-2017-17855 Improper use of pointers in place of scalars. https://nvd.nist.gov/vuln/detail/CVE-2017-17855. (2017). Accessed: 2025-05-05.

[3] 2017. CVE-2017-17862 Improper branch-pruning logic. https://nvd.nist.gov/vuln/detail/CVE-2017-17862. (2017). Accessed: 2025-05-05.

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

[5] Daniel Borkmann. 2023. bpf: Fix __reg_bound_offset 64->32 var_off subreg propagation. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=7be14c1c9030f73cc18b4ff23b78a0a081f16188. (2023). Accessed: 2025-05-05.

[6] Patrick Cousot and Radhia Cousot. 1977. 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)*. Association for Computing Machinery, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

[7] Leonardo De Moura and Nikolaj Bjørner. 2008. 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 (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[8] John Fastabend. 2020. bpf: Verifer, adjust_scalar_min_max_vals to always call update_reg_bounds(). https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=294f2fc6da27620a506e6c050241655459ccd6bd. (2020). Accessed: 2025-05-05.

[9] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. Component-based synthesis for complex APIs. *SIGPLAN Not.* 52, 1 (Jan. 2017), 599–612. https://doi.org/10.1145/3093333.3009851

[10] Google. 2023. Buzzer - An eBPF Fuzzer toolchain. https://github.com/google/buzzer. (2023). Accessed: 2025-05-05.

[11] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 62–73. https://doi.org/10.1145/1993498.1993506

[12] Hsin-Wei Hung and Ardalan Amiri Sani. 2024. BRF: Fuzzing the eBPF Runtime. *Proc. ACM Softw. Eng.* 1, FSE, Article 52 (July 2024), 20 pages. https://doi.org/10.1145/3643778

[13] Ben Hutchings. 2017. bpf/verifier: Fix states_equal() comparison of pointer and UNKNOWN. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=37435f7e80ef9adc32a69013c18f135e3f434244. (2017). Accessed: 2025-05-05.

[14] Juan Jose. 2023. Linux Kernel: Vulnerability in the eBPF verifier register limit tracking. https://github.com/google/security-research/security/advisories/GHSA-hfqc-63c7-rj9f. (2023). Accessed: 2025-05-05.

[15] Juan Jose, Meador Inge, Simon Scannell, and Valentina Palmiotti. 2023. Path pruning gone wrong. https://github.com/google/security-research/security/advisories/GHSA-j87x-j6mh-mv8v. (2023).

[16] Pankaj Kumar Kalita, Sujit Kumar Muduli, Loris D'Antoni, Thomas Reps, and Subhajit Roy. 2022. Synthesizing abstract transformers. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 171 (Oct. 2022), 29 pages. https://doi.org/10.1145/3563334

[17] Pankaj Kumar Kalita, Thomas Reps, and Subhajit Roy. 2025. Automated Abstract Transformer Synthesis for Reduced Product Domains. *ACM Trans. Softw. Eng. Methodol.* (May 2025). https://doi.org/10.1145/3733716 Just Accepted.

[18] Lucas Leong. 2021. CVE-2021-31440: An Incorrect Bounds Calculation in the Linux Kernel eBPF Verifier. https://www.zerodayinitiative.com/blog/2021/5/26/cve-2021-31440-an-incorrect-bounds-calculation-in-the-linux-kernel-ebpf-verifier. (2021).

[19] Chu Min Li and Felip Manya. 2021. MaxSAT, hard and soft constraints. In *Handbook of satisfiability*. IOS Press, 903–927.

[20] Linux Kernel Community. 2024. eBPF In-Kernel Test Suite (selftests/bpf). https://elixir.bootlin.com/linux/v6.14/source/tools/testing/selftests/bpf. (2024). Accessed: 2025-05-16.

[21] Linux Kernel Community. 2024. eBPF verifier operator for addition on unsigned intervals. https://elixir.bootlin.com/linux/v6.14/source/kernel/bpf/verifier.c#L14123. (2024). Accessed: 2025-05-16.

[22] Linux Kernel Community. 2024. eBPF verifier operator for subtraction on unsigned intervals. https://elixir.bootlin.com/linux/v6.14/source/kernel/bpf/verifier.c#L14168. (2024). Accessed: 2025-05-16.

[23] Andrii Nakryiko. 2023. bpf register bounds logic and testing improvements. https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf-next.git/commit/?id=cd9c127069c040d6b022f1ff32fed4b52b9a4017. (2023). Accessed: 2025-05-05.

[24] Valentina Palmiotti. 2021. Kernel pwning with eBPF: a love story. https://chomp.ie/Blog+Posts/Kernel+Pwning+with+eBPF+-+a+Love+Story. (2021).

[25] Manfred Paul. 2020. 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. (2020).

[26] Thomas Reps, Mooly Sagiv, and Greta Yorsh. 2004. Symbolic Implementation of the Best Transformer. In *Verification, Model Checking, and Abstract Interpretation*. Springer, Springer Berlin Heidelberg, Berlin, Heidelberg, 252–266.

[27] Thomas Reps and Aditya Thakur. 2016. Automating abstract interpretation. In *Verification, Model Checking, and Abstract Interpretation*. Springer, Springer Berlin Heidelberg, Berlin, Heidelberg, 3–40.

[28] Matan Shachnai. 2024. bpf, verifier: Improve precision of BPF_MUL. https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf-next.git/commit/?id=9aa0ebde0014. (2024). Accessed: 2025-05-05.

[29] Matan Shachnai, Harishankar Vishwanathan, Srinivas Narayana, and Santosh Nagarakatte. 2024. Fixing Latent Unsound Abstract Operators in the eBPF Verifier of the Linux Kernel. In *Static Analysis*. Springer Nature Switzerland, Cham, 386–406.

[30] Armando Solar-Lezama. 2008. *Program synthesis by sketching*. University of California, Berkeley.

[31] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 404–415. https://doi.org/10.1145/1168857.1168907

[32] Yonghong Song. 2020. bpf: fix a verifier failure with xor. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=2921c90d4718. (2020). Accessed: 2025-05-05.

[33] Hao Sun and Zhendong Su. 2024. Validating the eBPF Verifier via State Embedding. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 615–628. https://www.usenix.org/conference/osdi24/presentation/sun-hao

[34] Hao Sun, Yiru Xu, Jianzhong Liu, Yuheng Shen, Nan Guan, and Yu Jiang. 2024. Finding Correctness Bugs in eBPF Verifier with Structured and Sanitized Program. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 689–703. https://doi.org/10.1145/3627703.3629562

[35] Harishankar Vishwanathan. 2025. bpf, verifier: Improve precision for BPF_ADD and BPF_SUB. https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf-next.git/commit/?id=7a998a731627. (2025). Accessed: 2025-06-24.

[36] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2022. Sound, Precise, and Fast Abstract Interpretation with Tristate Numbers. In *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization (CGO '22)*. IEEE Press, 254–265. https://doi.org/10.1109/CGO53902.2022.9741267

[37] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2023. 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*. Springer-Verlag, Berlin, Heidelberg, 226–251. https://doi.org/10.1007/978-3-031-37709-9_12

[38] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2025. Vayu: Synthesizing Precise Abstract Operators for the eBPF Verifier. https://github.com/bpfverif/vayu. (2025). Accessed: 2025-06-24.

[39] Shung-Hsi Yu. 2024. bpf, verifier: improve signed ranges reasoning . https://lore.kernel.org/bpf/20240719110059.797546-6-xukuohai@huaweicloud.com/. (2024). Accessed: 2025-05-05.