Fixing Latent Unsound Abstract Operators in the eBPF Verifier of the Linux Kernel

Matan Shachnai⋆, Harishankar Vishwanathan⋆, Srinivas Narayana, and Santosh Nagarakatte

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

Abstract. This paper describes our experience deploying automated verification techniques for proving the correctness of value tracking components of the eBPF verifier in the Linux Kernel over the last four years. The eBPF verifier uses abstract interpretation with multiple abstract domains for value tracking. The eBPF verifier uses non-standard approaches for refining the results from multiple abstract domains, which necessitated us to design new techniques to show their correctness. During this process, we also discovered that some of the abstract operators are unsound in isolation. The unsoundness of these operators are eventually corrected by a shared refinement operator. The presence of intermediate "latent" unsound abstract operators makes the task of verification harder. We describe our patches to the Linux kernel, which have been upstreamed, that fix these latent errors and make the abstract operators correct in isolation, which enables faster automated verification.

Keywords: Abstract interpretation · Kernel extensions · eBPF

1 Introduction

Extending the functionality of the Linux kernel with user-augmented functionality is necessary in many contexts including cloud-native environments for observability, security [15, 37], telemetry, and load balancing [14]. The Extended Berkeley Packet Filter (eBPF) ecosystem is a collection of tools and techniques to extend the functionality of the Linux operating system kernel with safety assurances. Specifically, the eBPF ecosystem consists of a domain-specific language and an in-kernel register virtual machine with a 64-bit instruction set. A salient feature of this ecosystem is the eBPF verifier. The eBPF verifier is a static analyzer that checks whether a program is safe to execute within the Linux kernel using *abstract interpretation*. The eBPF verifier checks that the program terminates after executing a finite number of instructions, accesses to the memory locations are safe, and the program only accesses a subset of kernel memory and functions. Once the program is deemed safe by the verifier, the eBPF programs

⋆ Equal contribution.

are Just-in-Time (JIT) compiled to the native machine. Today, there are many applications and companies that use eBPF to instrument the Linux operating system running on production systems, implementing novel features for networking, storage, security, and performance monitoring [6, 14–16, 20, 37, 39, 55, 57].

The eBPF verifier employs abstract interpretation [33] to reason about program safety. Arguably, the eBPF verifier is the world's most widely used abstract interpreter, running on billions of devices worldwide. Unlike typical uses of abstract interpretation where the analysis is done in an offline setting, the eBPF verifier performs abstract interpretation in a live production setting. Hence, along with correctness of the analysis, its performance is also extremely important. The abstract interpreter, the accompanying abstract domains, and the various algorithms are implemented in an efficient manner in the eBPF verifier. Given that the eBPF verifier is executed in a production kernel, any bug in the verifier creates a huge attack surface for exploits [43, 44, 52, 54] and vulnerabilities [2, 5, 7–13, 24–27, 35, 40–42].

The core components of the eBPF verifier are the mechanisms used to track the values of program variables which are subsequently used to access memory. The eBPF verifier uses five abstract domains to track the values of variables (i.e., value tracking). Four of them are variants of the interval domain. The other is a bitwise domain named `tnum` [45, 47, 53, 61]. The eBPF verifier implements abstract operators for each of these domains efficiently. Conventionally, results from multiple abstract domains are combined using sound composition of sound abstract operators using modular reduced products [31]. The eBPF verifier in the Linux kernel combines the results from the various abstract domains in a non-modular fashion; it mixes up the implementation of abstract operators in one domain with reduction operators that combine information across domains [63]. The Linux kernel developers previously did not provide any soundness guarantees for these operators in the eBPF verifier.

Our efforts to push automated formal methods to verify the Linux kernel's eBPF verifier. Over the last few years, we have been using automated verification methods to check the correctness of various individual abstract domains and their composition in the eBPF verifier. Initially, we formalized the `tnum` abstract domain [61] which is efficiently implemented with fast operations in the Linux eBPF verifier. We also proposed a new abstract multiplication algorithm that is provably sound and is faster than previous algorithms [61]. The Linux kernel developers were more interested in the formal proofs of correctness when compared to performance from our new algorithm. Our algorithm is now incorporated in the Linux kernel since v5.14 [19].

Subsequently, we focused our efforts on proving the correctness of abstract interpretation algorithms for the entire value tracking analysis that includes the combination of the `tnum` domain and the interval domains. Our prototype, Agni [63], automatically checks the soundness of value tracking performed by the eBPF verifier. Agni automatically generates logical formulae representing the semantics of the abstract operator from eBPF verifier's C code (instead of manually writing them). We developed the correctness specification for value tracking given that eBPF verifier combines abstract operators for individual domains with refinement operations that combine information across domains. We used the Agni prototype [22, 62] to automatically check the soundness of 16 kernel versions starting from v4.14 to v5.19. During this process, we discovered 27 previously unknown bugs, which have been subsequently fixed by unrelated patches.

Of particular note were six abstract operators (corresponding to the instructions `bpf_and`, `bpf_and_32`, `bpf_or`, `bpf_or_32`, `bpf_xor`, `bpf_xor_32`) in the latest version of the eBPF verifier (i.e., v5.19 when Agni was initially released) that we found to be unsound. We observed that the eBPF verifier, in addition to performing non-modular refinements, uses a *shared refinement operator*, at the tail end of every abstract operator. This shared operator effectively preconditions input abstract values to all abstract operators. As a result, this prevents soundness issues from being manifested by any concrete eBPF program. However, we were still concerned that this "latent" unsoundness can cause potential issues in the future with changes to the shared refinement operator. Finally, the issue of latent unsoundness could not be overlooked when we could not verify the soundness of the abstract operators in the latest Linux kernels (v6.3 or later). A few kernel developers have been using the Agni prototype to check the correctness of the latest commits pertaining to value tracking [1, 17, 18]. Starting around v6.3, commits that make significant changes to the shared refinement operator caused our Agni prototype to take a very long time to complete verification (e.g., verification time increased to weeks from a magnitude of hours) and would not even complete for the latest version of the kernel (v6.8).

Novel contributions in this paper. To address the runtime issue, we want to split the task of verifying the entire abstract operator into smaller tasks and then compose them. To compose these smaller verification tasks, it is necessary to remove the latent unsoundness. Hence, we analyze the counterexamples generated by Agni and develop patches that remove the latent unsoundness from the six operators. By fixing the latent unsoundness, we are able to split the verification task, effectively achieving significant speedup in the verification process. We are now able to verify all the latest kernels including the latest version v6.8. The patches that make the six abstract operators sound have been accepted by the Linux kernel developers in the `bpf-next` kernel tree and are scheduled to be upstreamed to `mainline`.

Overall, this paper makes the following contributions.

1. We design a divide-and-conquer verification approach that allows us to scale verification to more complex refinement operators in the latest kernels (§3). Unlike [63], this approach does not suffer verification timeouts.
2. We investigate and report on the fundamental reasons for the latent unsoundness in the eBPF verifier's abstract operators. We propose fixes to make the abstract operators sound. These fixes have been accepted by kernel developers and have made their way into mainline Linux kernels (§4).
3. To support the significant code changes made to the eBPF verifier's refinement operators since v6.3, we re-engineered Agni in collaboration with the

kernel developers. This involved supporting new LLVM constructs in translating the C source code to verification conditions (§5).
4. Overall, this paper offers a case study on the effort (conceptual and engineering) required to push formal methods research to practical settings (§6).

2 Background

We first describe the various abstract domains along with the abstract operators that the eBPF verifier uses for value tracking. Next, we describe the soundness specification that Agni uses to verify the soundness of the abstract operators. Finally, we describe our experience with verifying changing abstract operators across several kernel versions.

2.1 Abstract Interpretation in the eBPF Verifier

The eBPF verifier uses five abstract domains for value tracking: four of them are interval domains (i.e., unsigned 64-bit (`u64`), unsigned 32-bit (`u32`), signed 64-bit (`s64`), signed 32-bit (`s32`)), and the fifth is a bitwise domain (called the tristate number, or `tnum` domain [51, 61]). Although the eBPF verifier is extensively tested, there was no formal specification for either the abstract domains or the operators before our prior work [61, 63].

The interval domains. The `u64` abstract domain is an interval domain that tracks an upper and lower bound of a 64-bit register when interpreted as an unsigned 64-bit value, across executions of the eBPF program. The `u64` abstract domain formally is $\mathbb{A}_{u64} \triangleq \{[x, y] \mid (x, y \in \mathbb{Z}^{+}_{64}) \wedge (x \leq_{u64} y)\}$, where $\mathbb{Z}^{+}_{64}$ is the set of 64-bit non-negative integers, and $\leq_{u64}$ represents a 64-bit unsigned comparison. The C code of the eBPF verifier maintains a tuple of unsigned 64-bit integers (`u64_min`, `u64_max`) for tracking the upper and lower bound of each register that appears in the eBPF program. The *concretization function* is $\gamma_{u64}([x, y]) \triangleq \{z \mid (z \in \mathbb{Z}^{+}_{64}) \wedge (x \leq_{u64} z \leq_{u64} y)\}$. The *abstraction function* is $\alpha_{u64}(c) \triangleq [min_{u64}(c), max_{u64}(c)]$, where c is a member of the powerset of $\mathbb{Z}^{+}_{64}$, and $min_{u64}(\cdot)$ and $max_{u64}(\cdot)$ compute the minimum and maximum over a finite set c where each element of c is interpreted as a 64-bit unsigned value. The other three interval domains, the signed 64-bit domain (`s64`), the unsigned 32-bit domain (`u32`), and the signed 32-bit domain (`s32`) are similarly implemented using the corresponding signed or unsigned arithmetic over the respective bitwidth.

```
def abstract_u64_add (in1, in2):
  if (in1.u64_min + in2.u64_min < in2.u64_min ||
      in1.u64_max + in2.u64_max < in2.u64_max):
    out.u64_min = 0;
    out.u64_max = UINT64_MAX;
  else
    out.u64_min = in1.u64_min + in2.u64_min
    out.u64_max = in1.u64_max + in2.u64_max
  return out
```

Fig. 1: Abstract addition in the `u64` domain on a `bpf_add` operation. Here, `in1` and `in2` are the abstract state maintained for two operands input to `bpf_add`. `UINT64_MAX` is the largest representable unsigned 64-bit integer.

An *abstract operator* captures the computation of concrete operations over program variables in the abstract domain. Figure 1 provides the pseudo code in the eBPF verifier for abstract `u64` addition. It has two input abstract states `in1` and `in2` corresponding to the inputs to the `bpf_add` instruction. It checks if the addition operation causes integer overflow, then sets the resulting bounds to the set of all integers in the `u64` domain (i.e., it loses precision). Otherwise, the bounds are updated as shown in the figure similar to interval arithmetic [32].

The tristate numbers (`tnum`) domain. This abstract domain in the eBPF verifier is similar to bitwise domains in abstract interpretation literature [45, 47, 53]. The goal of this domain is to track whether a bit of a given register is a known `0`, a known `1` or unknown across executions of the program. The eBPF verifier implements this domain with a tuple of two unsigned 64-bit integers (`v`,`m`). If a particular bit of `m` is 1 then the value of that bit of the register is unknown. If a particular bit of `m` is 0 then the value of that bit of the register is equal to `v`'s value for that particular bit.

Combining information from multiple domains. The eBPF verifier implements abstract operators for each abstract domain corresponding to each arithmetic and logic (ALU) instruction and each jump instruction in the eBPF instruction set. Consider two abstract domains with sets of abstract values $\mathbb{A}_1, \mathbb{A}_2$. For a fixed concrete operator $f\colon \mathbb{C} \to \mathbb{C}$, suppose the abstractions of f in the two domains respectively are g_1 and g_2 ($g_1\colon \mathbb{A}_1 \to \mathbb{A}_1$, and $g_2\colon \mathbb{A}_2 \to \mathbb{A}_2$) Depending on f, the precisions of g_1 and g_2 may vary significantly. Therefore, the benefit of using multiple abstract domains is that it allows combining information from different domains which may improve precision.

Intuitively, the eBPF verifier aims to make the abstract value in one domain more precise using information available in an abstract value in a different domain. This is typically done by using a separate refinement operator [34]. The eBPF verifier combines both the abstract operator and refinement steps. Consider the following refinement that happens in the abstract operator corresponding to `bpf_and` in the eBPF verifier. Here, `in1` and `in2` are input abstract *states* that encapsulate all the five domains, and `out` is the output abstract state that is being calculated. Prior to this snippet, using information purely from the input tnum domains, `out.tnum.v` has already been calculated.

```
out.u64_min = out.tnum.v;
out.u64_max = min(in1.u64_max, in2.u64_max);
```

In this abstract operator for the `u64` domain, the lower bound, `u64_min`, is computed using the *output* `tnum` information (i.e., `out.tnum.v`), which is a refinement operation. The upper bound, `u64_max`, is computed using the *input* `u64` information, which is a traditional abstraction operation. Hence, an abstract operator in the eBPF verifier performs refinement along with the abstract operation. Hence, reasoning about the correctness of these abstract operators using traditional methods prevalent in the literature such as modular reduced products is not possible.

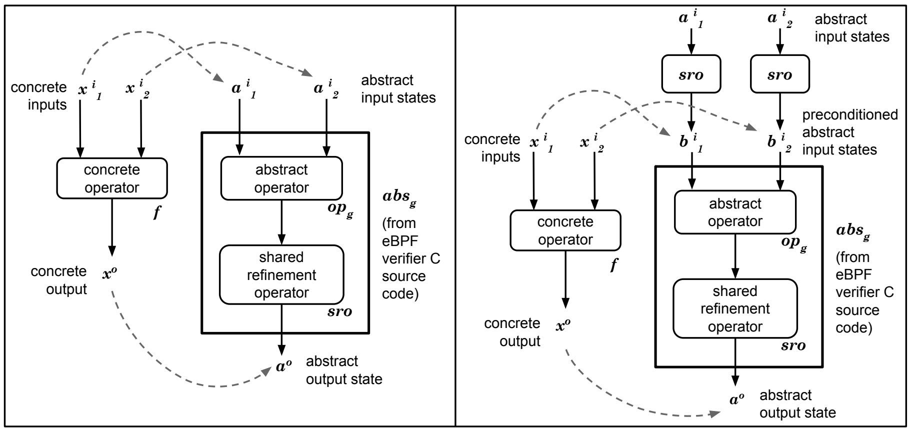

Fig. 2: Our approach to check both general soundness (left) and preconditioned soundness with the shared refinement operator (sro). Here, f is the concrete eBPF operation and abs_g is the the abstract operator from the source code in the Linux eBPF verifier. The dotted arrows indicate that a concrete value is a member of the abstract state (e.g., x^i_1 is a member of the abstract state a^i_1).

2.2 Soundness Specification for eBPF Verifier's Abstract Operators.

We provide a quick overview of our method to check soundness of an abstract operator. A detailed treatment can be found in our CAV paper [63]. Given a concrete eBPF operation $f\colon \mathbb{C} \times \mathbb{C} \to \mathbb{C}$ over the concrete domain $\mathbb{C}$ and an abstract operator $g\colon \mathbb{A} \times \mathbb{A} \to \mathbb{A}$ over abstract states $\mathbb{A}$, the operator g is sound if $\forall a_1, a_2 \in \mathbb{A} : f(\gamma(a_1), \gamma(a_2)) \sqsubseteq_{\mathbb{C}} \gamma(g(a_1, a_2))$.

We represent the fact that a concrete value $x \in \mathbb{C}$ is contained in the concretization of the abstract $a \in \mathbb{A}$ with the formula $mem_{\mathbb{A}}(x, a)$. For example for the s64 domain, $mem_{s64}(x, a) \triangleq (a.min \leq_{s64} x) \wedge (x \leq_{s64} a.max)$. The input-output relationship of the abstract operator obtained from the verifier's source code is represented as $a^o = abs_g(a^i_1, a^i_2)$, where a^i_1 and a^i_2 are input abstract values and a^o is the output abstract value. The abstract operator abs_g corresponding to the concrete operation f is sound when the formula in Equation 1 is valid.

$$\begin{aligned} \forall x^i_1,\ \ x^i_2 \in \mathbb{C},\ \ a^i_1, a^i_2 \in \mathbb{A} : mem_{\mathbb{A}}(x^i_1, a^i_1) \wedge mem_{\mathbb{A}}(x^i_2, a^i_2) \wedge \\ x^o = f(x^i_1, x^i_2) \wedge a^o = abs_g(a^i_1, a^i_2) \Rightarrow mem_{\mathbb{A}}(x^o, a^o) \end{aligned} \tag{1}$$

We adapt the above soundness condition to account for five abstract domains used by the eBPF verifier. We demonstrate it with two abstract domains $\mathbb{A}_1$ and $\mathbb{A}_2$ where abstract values a^i_{11} and a^i_{21} are in domain $\mathbb{A}_1$, and abstract values a^i_{12} and a^i_{22} are in domain $\mathbb{A}_2$. The concrete input x^i_1 must be contained in the concretization of the abstract values in domain $\mathbb{A}_1$. Hence, we assert

$mem_{\mathbb{A}_1}(x^i_1, a^i_{11}) \wedge mem_{\mathbb{A}_2}(x^i_1, a^i_{12})$. We apply the same reasoning for x^i_2. The input-output relationship of the abstract operator from the eBPF verifier source code for two domains can be represented as $\{a^o_1, a^o_2\} = abs_g(a^i_{11}, a^i_{12}, a^i_{21}, a^i_{22})$. The abstract operator is sound if the concrete output is a subset of the concretizations of the abstract outputs in each domain, i.e., $mem_{\mathbb{A}_1}(x^o, a^o_1) \wedge mem_{\mathbb{A}_2}(x^o, a^o_2)$. The formula to check soundness with two abstract domains is shown below in Equation 2.

$$\begin{aligned}
\forall x^i_1,\ \ x^i_2 \in \mathbb{C},\ \ a^i_{11},\ \ a^i_{21} \in \mathbb{A}_1,\ a^i_{12},\ a^i_{22} \in \mathbb{A}_2 :\\
mem_{\mathbb{A}_1}(x^i_1, a^i_{11}) \wedge mem_{\mathbb{A}_2}(x^i_1, a^i_{12}) \wedge mem_{\mathbb{A}_1}(x^i_2, a^i_{21}) \wedge mem_{\mathbb{A}_2}(x^i_2, a^i_{22}) \wedge\\
x^o = f(x^i_1, x^i_2) \wedge \{a^o_1, a^o_2\} = abs_g(a^i_{11}, a^i_{12}, a^i_{21}, a^i_{22})\\
\Rightarrow (mem_{\mathbb{A}_1}(x^o, a^o_1) \wedge mem_{\mathbb{A}_2}(x^o, a^o_2))
\end{aligned} \tag{2}$$

2.3 eBPF Verifier's Input Preconditioning.

On checking various versions of the Linux eBPF verifier with the above specification, our Agni prototype discovered that some of the abstract operators are indeed unsound. To get the attention of kernel developers, we had to determine if this unsoundness can actually manifest with any concrete eBPF program. During this exploration, we discovered that every abstract operator in the eBPF verifier performs a *shared suffix* of refinement operations at the end (see Figure 2). The purpose of this *shared refinement operator* is to combine information from all abstract domains. The encoding abs_g obtained from the source code already includes this shared refinement operator.

We discovered that the soundness specification above allowed any valid input abstract state for the abstract operator. In contrast, the input abstract state for any abstract operator is either the initial state (i.e., any concrete value or a singleton known concrete value) or the output abstract state produced by another abstract operator, which is preconditioned by the shared refinement operator! Hence, we refined our soundness specification to precondition the input abstract states based on this shared refinement operator (see [63]).

2.4 Experience Checking Various Kernel Versions.

Using the preconditioned soundness specification, we were able to check the soundness of 23 versions of the eBPF verifier starting from v4.14 to v6.3. In this process, we rediscovered numerous bugs, which were known to the developers with documented CVEs. Some of them were already fixed by the kernel developers accidentally in unrelated patches. Table 2 reports the time taken for verifying all abstract operators on average for various kernel versions.

Until kernel v5.19, we could check the soundness of these operators in a few hours. The verification time starting from v5.19 increased to 36 hours. Starting from v6.4, our queries to the SMT solver would timeout after running for a few

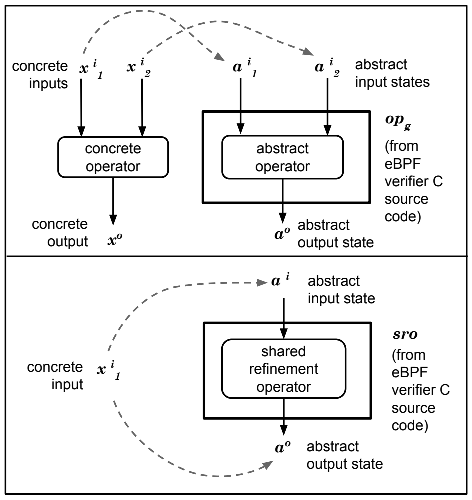

Fig. 3: Our divide and conquer approach decouples the abstract operator ($\langle op_g \rangle$) and the shared refinement operator ($\langle sro \rangle$) into two separate verification conditions which we can verify, both are extracted from kernel source code. Since the shared refinement does not relate any concrete operation, its concrete input and output is the same. The dotted arrow indicates that the concrete value is a member of the abstract state (e.g., x^i_1 is a member of the abstract state a^i_1).

weeks. We observed that there were multiple updates to the shared refined operator, which increased its complexity, and is the likely cause of timeouts. While many of our verification queries resulted in timeouts (i.e. abstract operators for `bpf_sub` and `bpf_and`), we did find that six abstract operators—`bpf_and`, `bpf_or`, `bpf_xor`, `bpf_and_32`, `bpf_or_32`, and `bpf_xor_32`—were unsound using our general soundness specification.

At this point we faced two significant problems; (1) our verification methodology was too slow to see wider adoption, and (2) some of these abstract operators were unsound when using our general soundness specification. To tackle the first problem, we wanted to explore a divide and conquer strategy that would allow us to split the abstract operator into smaller parts which we could verify.

3 A Divide and Conquer Approach for Verification

Intuitively, an abstract operator in the verifier can be considered as a composition of two sub-operators, $\langle abs_g \rangle = \langle op_g \rangle \cdot \langle sro \rangle$, where $\langle op_g \rangle$ is executed followed by $\langle sro \rangle$. Here $\langle op_g \rangle$ represents the unique part of the abstract operator that updates abstract domains according to the specific operator (i.e. addition, multiplication, bitwise-and, bitwise-right-shift, etc.) and $\langle sro \rangle$ represents the refinement operator that is shared across all abstract operators (i.e. the shared refinement operator). Starting from v6.4, our verification paradigm [63]—generating a single large verification condition for $\langle op_g \rangle \cdot \langle sro \rangle$—was no longer viable since it would take weeks for the verification process to finish. Hence we needed a new approach for faster verification.

The compositional nature of abstract operators in the verifier presented an opportunity for quicker verification. Our key insight here, depicted in Figure 3, is that we could decouple $\langle op_g \rangle$ and $\langle sro \rangle$ and generate verification conditions for them, then prove their correctness separately.

Soundness specification for $\langle op_g \rangle$**.** Our primary strategy for proving the correctness of the latest abstract operators in the eBPF verifier is to split the task into smaller subtasks and individually verify their correctness. The soundness specification for op_g is similar to the one in Equation 2, except that abs_g is replaced by op_g such that $\{a_1^o, a_2^o\} = op_g(a_{11}^i, a_{12}^i, a_{21}^i, a_{22}^i)$. The formula in Equation 3 below illustrates this minor change for two abstract domains:

$$\begin{aligned} \forall x_1^i,\ \ x_2^i \in \mathbb{C},\ \ a_{11}^i,\ \ a_{21}^i \in \mathbb{A}_1,\ a_{12}^i,\ a_{22}^i \in \mathbb{A}_2 : \\ mem_{\mathbb{A}_1}(x_1^i, a_{11}^i) \wedge mem_{\mathbb{A}_2}(x_1^i, a_{12}^i) \wedge mem_{\mathbb{A}_1}(x_2^i, a_{21}^i) \wedge mem_{\mathbb{A}_2}(x_2^i, a_{22}^i) \wedge \\ x^o = f(x_1^i, x_2^i) \wedge \{a_1^o, a_2^o\} = op_g(a_{11}^i, a_{12}^i, a_{21}^i, a_{22}^i) \\ \Rightarrow (mem_{\mathbb{A}_1}(x^o, a_1^o) \wedge mem_{\mathbb{A}_2}(x^o, a_2^o)) \end{aligned} \tag{3}$$

Soundness specification for $\langle sro \rangle$**.** We need to define what it means for the shared refinement operator (i.e. sro) to be sound. We use $sro(a)$ to denote the abstract value that is the output of the shared refinement operator which we extract from verifier source code. Note that $sro(a)$ takes a single abstract input and produces a single abstract output. Additionally, in contrast to an abstract operator that has a corresponding concrete eBPF operator, the shared refinement operator is not related to any concrete eBPF operation. Hence, we use an identity function as the concrete operator in our soundness specification for sro. Given concerete input x^i that is contained in the concretization of abstract inputs a_1^i and a_2^i, we say that shared refinement operator is sound if the concrete output x^o is contained within the concretization of resulting abstract outputs a_1^o and a_2^o. The formula to check the soundness of the shared refinement operator that refines the output abstract state based on two abstract domains is shown below in Equation 4.

$$\begin{aligned} \forall x^i \in \mathbb{C},\ \ a_1^i\ \in \mathbb{A}_1,\ a_2^i\ \in \mathbb{A}_2 : mem_{\mathbb{A}_1}(x^i, a_1^i) \wedge mem_{\mathbb{A}_2}(x^i, a_2^i) \wedge \\ x^o = x^i \wedge \{a_1^o, a_2^o\} = sro(a_1^i, a_2^i) \Rightarrow (mem_{\mathbb{A}_1}(x^o, a_1^o) \wedge mem_{\mathbb{A}_2}(x^o, a_2^o)) \end{aligned} \tag{4}$$

Overall, our divide and conquer verification approach produces smaller SMT formulae for every abstract operator abs_g, and the single refinement operator sro. Consequently, these formulae can be verified in a matter of minutes. However, it requires each unique abstract operator op_g to be sound so that we can reason about the soundness of its composition with the refinement operator. Hence, we started exploring why some operators were unsound and developed fixes that correct these cases of unsoundness.

4 Making the eBPF Verifier's Abstract Operators Sound

Now, our goal is to fix the unsound abstract operators, which will allow us to individually check the soundness of the abstract operators separately from

```
case bpf_and:
  out.tnum = tnum_and(in1, in2);
  out.s32, out.u32 = interval_and_32(in1, in2);
  out.s64, out.u64 = interval_and_64(in1, in2);
```

(a)

```
1  def interval_and_64(in1, in2):
2  # ...
3      out.u64_min = in1.tnum.v;
4      out.u64_max = min(in1.u64_max,
5                        in2.u64_max);
6  if (in1.s64_min < 0 ||
7      in2.s64_min < 0):
8      out.s64_min = INT64_MIN;
9      out.s64_max = INT64_MAX;
10 else:
11     out.s64_min = out.u64_min;
12     out.s64_max = out.u64_max;
13 # ...
```

(b)

Fig. 4: (a) An illustration of the sequence of calls that update the various abstract domains. (b) The specific sequence of updates to the `u64` and `s64` domain that leads to unsoundness in the abstract operator. Specifically, the signed domain is updated by using the unsigned domain. However, the check on line 6 does not guarantee that implicit casts from a 64-bit unsigned value to a 64-bit signed value (lines 11 and 12) do not result in integer overflows.

the shared refinement operator. We use the Agni prototype [62] to just check the abstract operator without the shared refinement operator with the general soundness specification shown in Figure 2. Specifically, the prototype reports that the bitwise operators, `bpf_and`, `bpf_or`, `bpf_xor` and their 32-bit counterparts `bpf_and_32`, `bpf_or_32`, and `bpf_xor_32` as unsound with the general soundness specification. Agni's models from SMT verification show that the output abstract values for the `s64` and `s32` domains can be illformed (i.e. `s64_min > s64_max` and `s32_min > s32_max`).

Figure 4a specifies the structure of the bitwise `and` abstract operation in the eBPF verifier; the abstract operator updates all five abstract domains by first updating its `tnum` domain using the respective `tnum` operation, then its 32-bit interval domains and finally, its 64-bit interval domains. This structure is important because the interval domains use the updated `tnum` domain to infer new bounds, which is the precursor for unsoundness in the operator.

Figure 4b shows how the interval domains are updated in the eBPF verifier for the `bpf_and` operator; first the lower bound of the unsigned 64-bit domain is inferred from the value in the `tnum` domain (line 3). Then, the upper bound of the output in the `u64` domain (line 4) is inferred using the upper bounds of the operands in the `u64` domain. Finally, the abstract state for the `s64` domain is inferred based on the updated state from the `u64` domain (lines 8-9 and 11-12). When the lower bounds of the operands in the `s64` domain are negative (line 6), then the verifier sets the bounds to the entire range of the `s64` domain, which loses all precision (lines 8 and 9). This condition ensures that signed bounds are inferred from the unsigned bounds only if both registers take on positive values (lines 11 and 12). However, this check is not correct and does not account for potential signed integer overflows that may happen in lines 11 and 12.

The model from our Agni prototype indicates that the resulting abstract state has unsound signed bounds (i.e., `s64_min > s64_max`). To reach such a state, the input `s64` abstract states should be positive which ensures that the else branch is taken for the condition in line 6. Given that the `u64` bounds are sound, `u64_min` has to be greater than `u64_max`. The range of values represented by an unsigned 64-bit value is larger than a signed 64-bit value. Hence, the unsoundness occurs when the most-significant bit (MSB) of `u64_max` is 1 (i.e., `u64_max` is greater than or equal to 2^{63}) and the MSB of `u64_min` is 0 (i.e., `u64_min` is less than or equal to $2^{63} - 1$). This snippet of code (line 6) is the root cause of the error.

There are three main cases that one needs to consider so that integer overflows do not occur when an unsigned 64-bit value is assigned to a signed 64-bit value. In the first case, $\texttt{u64\_min} \leq \texttt{u64\_max} \leq 2^{63}-1$. Both signed and unsigned values are identical and positive and within their respective dynamic ranges. The current check on line 6 in Figure 4b correctly handles this case. In the second case, $2^{63} - 1 < \texttt{u64\_min} \leq \texttt{u64\_max}$. Although the value in the unsigned representation exceeds the dynamic range of the signed 64-bit integer, both the lower bound and the upper bound in the resulting signed 64-bit representation will be negative values. The invariant `s64_min < s64_max` is still maintained. In the third case, $\texttt{u64\_min} \leq 2^{63} - 1 < \texttt{u64\_max}$. When these unsigned 64-bit bounds are assigned to signed 64-bit bounds, the `s64_min` bound will be positive (as `u64_min` is within the signed 64-bit value's dynamic range) and the `s64_max` will be negative. This case is not handled by the check in line 6 of Figure 4b.

Our patch to fix the unsoundness in the abstract operators. Our patch makes the abstract operator sound by correctly handling the above three cases. Specifically, our insight is to assign the unsigned 64-bit bounds to signed 64-bit bounds only when the invariant `s64_min <= s64_max` can be ensured. We replace lines 6-12 with the following snippet in Listing 1.1.

```
if ((s64) out.u64_min <= (s64) out.u64_max):
    out.s64_min = out.u64_min;
    out.s64_max = out.u64_max;
else:
    out.s64_min = INT64_MIN;
    out.s64_max = INT64_MAX;
```

Listing 1.1: Our patch to fix the unsoundness in six of the eBPF verifier's abstract operators

When we checked the fixed abstract operator with the Agni prototype, it reported that the patched operator is sound. Our corresponding patch that fixes the unsoundness of the eBPF's six abstract operators has been upstreamed to the kernel [3] [1].

[1] `https://go.rutgers.edu/90ueywub`

5 C to Logic—Supporting New LLVM Constructs

To generate verification conditions for the eBPF verifier's abstract operators, the first step in Agni involves extracting the semantics of an abstract operator. For each abstract operator, Agni automatically (a) converts the eBPF verifier's C code into LLVM IR, (b) obtains a slice of the eBPF verifier concerned with the particular abstract operator, and (c) converts the LLVM IR into logic in the SMT-LIB format using our LLVMToSMT compiler pass. Due to significant changes to the verifier's C code, the LLVMToSMT pass had to be re-engineered. This section discusses the details of these changes. We first begin with a refresher of the LLVMToSMT pass, and then proceed to discuss the LLVM new constructs that needed to be supported.

5.1 The LLVMToSMT Pass

The LLVMToSMT pass encodes the semantics of an abstract operator in the theory of bitvectors. The eBPF verifier's five abstract domains are encapsulated in a struct called `reg_st`. In general, each abstract operator takes as input two `reg_st` pointers (let's say `a` and `b`), and updates the memory pointed to by them. We model each `reg_st` as a tree (more simply, an array) of bitvectors. We create an array of bitvectors on function entry, corresponding to the input `reg_st` pointers `a` and `b`. This represents the view of memory on function entry (Figure 5).

```
liveOnEntry:
a: [a0, a1, a2, a3, a4]
b: [b0, b1, b2, b3, a4]
```

Fig. 5: Array of bitvectors for inputs `a` and `b` modeling the view of memory on function entry (bitvectors in blue).

Effectively, each LLVM instruction of the abstract operator utilizes the bitvectors from this array and generates formulas that uses them. On every LLVM instruction that creates a temporary register in IR, we create a fresh bitvector variable corresponding to that temporary register. For example, a `load` instruction `%ld1 = load i64, i64* %gep1` might be encoded using the fresh bitvector `ld1` as `(= ld1 b2)`, because a preceding `getelementptr` instruction calculated the address of `%gep1` as the 3^{rd} member of the input `reg_st b` (hence, `b2`). Most other instructions only operated on single value types e.g. `i32, i64`. Encoding them into formulas involves asserting that a fresh bitvector equals a combination of existing bitvectors based on the instruction's semantics. For example, a `select` instruction on single value types that looks like `%x1 = select i1 %cond, i1 %x2, i1 x3%`, is encoded as `ite (= cond #b1) (= x1 x2) (= x1 x3)`[2]. The `store` instructions create new views of memory, by modifying existing views of memory. LLVMToSMT leverages LLVM's MemorySSA [21] pass to figure out which memory view a `store` modifies. Finally, the memory view that is active when encountering the `ret` instruction contains the output bitvectors for the kernel's abstract operators.

[2] In SMT-LIB `#b1` is a bitvector of length 1 that is equal to the value 1

```
42  ...
43  %sel = select i1 %is_jmp32, %reg_st* %a, %reg_st* %b
44  %gep1 = getelementptr %reg_st, %reg_st* %sel, i64 0, i32 2
45  %ld1 = load i64, i32* %gep1
46  ...
47  %gep2 = getelementptr %reg_st, %reg_st* %sel, i64 0, i32 4
48  store i64 0, i64* %gep2
```

Listing 1.2: Code patterns in LLVM IR emitted by clang when compiling the newer versions of the eBPF verifier. The IR involves a `select` instruction on pointer types.

5.2 Supporting new LLVM constructs

Starting from kernel `v6.8-rc1` major changes were introduced to the register state bounds update logic [48], which resulted in LLVM IR code patterns that LLVMToSMT was not able to handle. We will consider the example of IR code patterns that involved `select` instructions on pointer types for the purposes of illustration. These instruction patterns were emitted by clang when compiling the eBPF verifier function `is_branch_taken` [4]. This function takes as input two `reg_st` pointers corresponding to the registers involved in a jump instruction and aims to determine statically if either the goto or the fall-through branch will always be taken at runtime. Importantly, this function involves conditionally swapping the two input `reg_st` pointers. When compiled to LLVM IR, clang emits the IR pattern in Listing 1.2 containing a `select` instruction that chooses between two `reg_st` pointers `a` and `b`.

Supporting* `loads` *from* `select` *on pointer types. Let's say the array of bitvectors for each input to the function `a` and `b` look like the one in Figure 5 on function entry. The `select` instruction (line `43`) determines *which* of the two input `reg_sts` is indexed by the subsequent `getelementptr` (line `44`), depending on the result of a condition (here, `is_jmp32`). That is, the bitvector corresponding to the load, `ld1`, should be either equal to `reg_st` `a`'s 3^{rd} bitvector a_2, or `reg_st` `b`'s 3^{rd} bitvector b_2. Thus, such a `load` instruction must be encoded with the following formula: `(ite (= is_jmp32 #b1) (= ld1` a_2`) (= ld1` b_2`))`.

Supporting* `stores` *from* `select` *on pointer types. A `store` instruction like the one on line `47` modifies the memory view. So, we first make a copy of the bitvectors in `liveOnEntry` to associate with new memory view that the `store` creates. Let's say this memory view is called `MemoryDef(1)`. The `store` will update this memory view. The `store` instruction stores a value at a memory location calculated by the preceding `getelementptr` instruction at line `44`, which in turn could be either at `reg_st` `a` or `reg_st` `b`, depending on the preceding `select` instruction. To handle this, we create new bitvectors `tempa` and `tempb` and store them at location 4 in our new bitvector arrays corresponding to `MemoryDef(1)`.

`MemoryDef(1):`
`a: [`a_0`,` a_1`,` a_2`,` a_3`, tempa]`
`b: [`b_0`,` b_1`,` b_2`,` b_3`, tempb]`

Now, additionally, we need a formula that asserts that the bitvectors `tempa` and `tempb` at the specific locations for either bitvector arrays are set according to the result of the boolean comparison that the `select` was based on (i.e.`is_jmp32`). That is, if the `is_jmp32` is true, `a[4]` will be updated, else it will remain unchanged. If `is_jmp32` is false, `b[4]` will be updated, else it will remain unchanged. The formula we obtain is:

```
(ite (= is_jmp32 #b1) (= tempa a₂) (= tempa #x0000000000000000))
(ite (= is_jmp32 #b0) (= tempb b₂) (= tempb #x0000000000000000))
```
[3]

Re-engineering LLVMToSMT. In addition to the above, the changes to the eBPF verifier in `v6.8-rc1`, required the handling of `phi` instructions in IR that choose between pointer types, (similar to `select` instructions on pointer types). Encoding such IR instructions requires tracking the dependencies between the `select`, `phi`, `getelementptr`, and `store` (and `load`) instructions, as illustrated above. This required significantly re-engineering LLVMToSMT in order to support newer versions of the abstract operators in the eBPF verifier.

6 Experience Verifying the Sound Patched Operators

We have significantly enhanced the Agni prototype [62] incorporating the feedback from the Linux kernel developers. A few kernel developers are actively using our prototype. We are actively working to integrate it as part of the CI process. To facilitate the possible integration into the CI process, we had to change our verification condition generator for the abstract operators directly from the eBPF verifier's C source code to account for new features used by the verifier. Further, we split the verification into smaller subtasks using a divide and conquer approach which we call the new strategy henceforth. With our new strategy we are able to verify that the latest abstract operators for value tracking in kernel v6.8 are sound.

Our patches make the abstract operators sound. We applied our fixes to both the latest kernel version (v6.8) and also to some of the older versions. Subsequently, we tested these versions of the eBPF verifier using the general soundness specification in Agni. Prior to our patch, bitwise operators (`bpf_and`/`bpf_or`/`bpf_xor` and their 32-bit counterparts) were unsound. Table 1 shows that our patch was able to fix the latent unsoundness in these abstract operators. Apart from these operators, other abstract operators were already sound even without the shared re-

eBPF Instruction	Before Patch?	After Patch?
`bpf_and`	✗	✓
`bpf_and_32`	✗	✓
`bpf_or`	✗	✓
`bpf_or_32`	✗	✓
`bpf_xor`	✗	✓
`bpf_xor_32`	✗	✓

Table 1: Verification results for latent unsound abstract operators in kernel v6.8 with and without our patch applied. After applying our patch these abstract operators become sound.

[3] In SMT-LIB `#x0000000000000000` is a bitvector of length 64 equal to the value 0

finement operator. Specifically our patches eliminate the need for doing `sro`-preconditioned verification, which significantly improves verification time.

Improvements in verification time with our new strategy. Our divide and conquer approach for verifying abstract operators in the eBPF verifier not only allowed us to prove that recent versions are indeed sound with respect to value tracking but it also significantly minimized verification runtime. Previously, when we tried to verify these kernel versions we would encounter timeouts. Table 2 reports the time taken to verify all the abstract operators in the eBPF verifier for value tracking. We conducted these experiments on the Cloudlab [36] framework, using two 10-core machines with Intel Skylake CPUs running at 2.2 GHz and 192GB of RAM.

Our old strategy that performed `sro`-preconditioned verification in the presence of latent unsound abstract operators was not able verify any kernel version starting from v6.4. Further, it took significant time on kernel versions before v6.3. Our new strategy is able to complete verification of all abstract operators in a given version in less than 15 minutes. This result is very useful in making a case for integration into the CI process of the eBPF verifier. Our new strategy, which requires no latent unsoundness in the abstract operators, is more robust to changes in the verifier's code. Kernel v6.8 introduced significant changes to the verifier's reduction operator and made it significantly larger. The Agni prototype using the old strategy was seeing timeouts. In contrast, the Agni prototype with the new strategy is able to verify all the operators of the latest eBPF verifier in less than 30 minutes, which also highlights the scalability and robustness of the new strategy in handling extensive changes to the eBPF verifier.

Kernel Version	Old Strategy	New Strategy
4.14	$\sim$2.5 hrs	$<$5 min
5.5	$\sim$2.5 hrs	$<$5 min
5.9	$\sim$4 hrs	$<$5 min
5.13	$\sim$10 hrs	$<$5 min
5.19	$\sim$36 hrs	$<$15 min
6.3	$\sim$36 hrs	$<$15 min
6.4	Timeout	$<$15 min
6.5	Timeout	$<$15 min
6.6	Timeout	$<$15 min
6.7	Timeout	$<$15 min
6.8	Timeout	$<$30 min

Table 2: Comparison of verification runtime performance between our old and new verification strategies. Times indicated are for verifying all instructions in a single kernel version.

Kernel developers are interested in using formal methods. Over the last four years working on this project, we are grateful to continuous encouragement and feedback from various Linux eBPF verifier developers. Without their feedback, we would not have been able to upstream our patches that proposed new algorithms for tnum multiplication [19] and made the abstract operators sound [3]. A few of them have also been using the Agni prototype [1, 17, 18]. Our experience suggests that real world adoption of formal method tools requires significant additional effort beyond the prototypes typically good enough for a research publication. In summary, we encourage the community to develop usable formal tools and collaborate with the eBPF verifier developers to push them into production for real-world impact.

7 Related Work

Our work uses and builds upon various seminal prior work on abstract interpretation [30, 31, 33]. Our soundness formalization is influenced by prior work on value tracking abstract domains [45,46,56]. When we tried to formalize the Linux eBPF verifier's method for combining abstract domains, we explored ideas by Cousot on enhancing the precision of abstract domains with reduced products and disjunctive completion domain refinements [29, 34], which has been later improved by others [60]. A systematic survey on product abstract operators is also available [28]. Our focus in this project has been to build on prior work so that we can easily apply them for automatic reasoning of the eBPF verifier in the Linux eBPF verifier.

This paper extends our previous work on Agni [63], which developed methods for automatically verifying the value tracking analysis in the eBPF verifier, introducing *sro* preconditioned verification. While our prior work checked the soundness of the eBPF verifier in Linux versions v4.14 to v5.19, it encountered verification timeouts in newer versions and did not address the causes of latent unsound abstract operators or propose fixes. In contrast, this paper introduces a faster, divide-and-conquer verification approach that eliminates the need for input preconditioning, investigates the causes of the latent unsoundness in the abstract operators, and proposes fixes. Additionally, we extend our C-to-logic framework to support new LLVM constructs, enabling the verification of abstract operators in the latest Linux kernel version, v6.8.

This paper, along with our initial Agni prototype [63], is closely related to the work by Bhat *et al.* [23], which also verifies value tracking in the eBPF verifier. However, Bhat *et al.* formalize a limited set of abstract operators and do not address the shared refinement operator or its underpinnings to soundness. Both their work and our previous work report latent unsound operators in the eBPF verifier. This paper, however, focuses on fixing latent unsound abstract operators to enable fast verification, with the aim of integrating formal methods into the continuous integration workflow of the Linux eBPF verifier.

In contrast to the Linux eBPF verifier's approach of eBPF verification in a live production kernel, the eBPF ecosystem in the Windows operating system performs offline verification using extensions of the PREVAIL verifier [38] in a secure user-mode environment. PREVAIL [38] uses relational zone abstract domains and can potentially verify richer eBPF primitives such as loops in comparison to the Linux eBPF verifier's verifier. It is unclear which of the two approaches, the offline approach of the Windows eBPF framework or the in-production verification in the Linux eBPF verifier, will become the dominant approach in the future.

Beyond formal verification, there are fuzzers employed to find bugs in the eBPF verifier. Recent work [58] generates structured eBPF programs that pass the verifier and then checks sanitized programs for bug indicators during runtime to discover potential verifier bugs. Beyond the verifier, there is significant work on formalizing and finding bugs in the JIT engine of the eBPF ecosystem [49, 50, 59, 64, 65], which is complementary to our work.

8 Conclusion

The Linux eBPF verifier uses abstract interpretation to verify the safety of input eBPF programs. Over the last four years, we have formalized various abstract domains, operators, and their soundness. We have automatically verified the soundness of the abstract operators by generating verification conditions directly from C code of the Linux eBPF verifier. We have upstreamed a new abstract operator for the tristate domain (`tnums`) and fixes to the latent unsoundness in six abstract operators in interval domains. Making our tool usable by Linux eBPF verifier developers through active collaboration has allowed them to explore the feasibility of integrating our tools as part of the eBPF verifier's continuous integration workflow. We hope that our experience encourages the formal methods community to collaborate with Linux eBPF verifier developers for real-world impact.

9 Acknowledgements

This paper is based upon work supported in part by the National Science Foundation under FMITF-Track I Grant No. 2019302 and the Facebook Systems and Networking Award. We thank the SAS reviewers and our shepherd, Sébastien Bardin, for their valuable feedback and guidance. We thank Paul Chaignon for his invaluable inputs and efforts to extend Agni. We also thank the kernel developers Andrii Nakryiko, Eduard Zingerman, Daniel Borkmann, and Alexei Starovoitov for their feedback to our patches. We also thank CloudLab for providing the research testbed for our experiments.

References

1. Agni's verification of kernel 6.4 takes weeks. https://github.com/bpfverif/agni/issues/13
2. bpf: fix incorrect sign extension in check_alu_op(). https://github.com/torvalds/linux/commit/95a762e2c8c942780948091f8f2a4f32fce1ac6f
3. 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
4. bpf, Register bounds logic and testing improvements. https://elixir.bootlin.com/linux/v6.8-rc1/source/kernel/bpf/verifier.c
5. bpf, x32: Fix bug with ALU64 LSH, RSH, ARSH BPF_X shift by 0. https://github.com/torvalds/linux/commit/68a8357ec15bdce55266e9fba8b8b3b8143fa7d2
6. Cilium API-aware networking and security. https://cilium.io/
7. CVE-2017-16996 Mishandling of register truncation. https://nvd.nist.gov/vuln/detail/CVE-2017-16996
8. CVE-2017-17852 Mishandling of 32-bit ALU ops. https://nvd.nist.gov/vuln/detail/CVE-2017-17852
9. CVE-2017-17853 Mishandling of 32-bit ALU ops. https://nvd.nist.gov/vuln/detail/CVE-2017-17853
10. CVE-2017-17864 Mishandled comparison between pointer and unknown data types. https://nvd.nist.gov/vuln/detail/CVE-2017-17864

11. CVE-2018-18445 Mishandling of 32-bit RSH op. https://nvd.nist.gov/vuln/detail/CVE-2018-18445
12. CVE-2020-8835 Mishandling of bounds tracking for 32-bit JMPs. https://nvd.nist.gov/vuln/detail/CVE-2020-8835
13. 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
14. Facebook's Katran load balancer: Kernel XDP program. https://github.com/facebookincubator/katran/blob/master/katran/lib/bpf/balancer_kern.c
15. Netconf 2018 day 1. https://lwn.net/Articles/757201/
16. Suricata: ebpf and xdp. https://suricata.readthedocs.io/en/latest/capture-hardware/ebpf-xdp.html
17. workflows: CI for the verification step . https://github.com/bpfverif/agni/commit/18b7513facda0d6f57e69c293f9a494ede683be7
18. workflows: Initial CI for the LLVM-to-SMT step . https://github.com/bpfverif/agni/commit/a30260d2b8c4c14f356f9501c4a9dac28f768f5d
19. 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)
20. New GKE Dataplane V2 increases security and visibility for containers. https://cloud.google.com/blog/products/containers-kubernetes/bringing-ebpf-and-cilium-to-google-kubernetes-engine (2021)
21. LLVM's MemorySSA. https://llvm.org/docs/MemorySSA.html (2023)
22. Verifying the Verifier: eBPF Range Analysis Verification (Apr 2023). https://doi.org/10.5281/zenodo.7931901
23. 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)
24. Borkmann, D.: bpf: Fix scalar32_min_max_or bounds tracking. https://github.com/torvalds/linux/commit/5b9fbeb75b6a98955f628e205ac26689bcb1383e (2020)
25. 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)
26. 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)
27. Borkmann, D.: bpf: Fix signed_sub,add32_overflows type handling. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=bc895e8b2a64e502fbba72748d59618272052a8b (2021)
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.: Abstract Interpretation Based Formal Methods and Future Challenges, pp. 138–156. Springer Berlin Heidelberg, Berlin, Heidelberg (2001). https://doi.org/10.1007/3-540-44577-3_10

31. 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 (2005)
32. 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)
33. 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
34. 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
35. 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)
36. Duplyakin, D., Ricci, R., Maricq, A., Wong, G., Duerig, J., Eide, E., Stoller, L., Hibler, M., Johnson, D., Webb, K., Akella, A., Wang, K., Ricart, G., Landweber, L., Elliott, C., Zink, M., Cecchet, E., Kar, S., Mishra, P.: The design and operation of cloudlab. In: Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference. p. 1–14. USENIX ATC '19, USENIX Association, USA (2019)
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. Gregg, B.: Bpf performance analysis at netflix. https://www.slideshare.net/brendangregg/reinvent-2019-bpf-performance-analysis-at-netflix
40. Horn, J.: Arbitrary read+write via incorrect range tracking in ebpf. https://bugs.chromium.org/p/project-zero/issues/detail?id=1454
41. 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)
42. 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)
43. 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
44. 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

45. 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
46. 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/10.1561/2500000034
47. 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
48. 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)
49. Nelson, L., Bornholt, J., Gu, R., Baumann, A., Torlak, E., Wang, X.: Scaling symbolic evaluation for automated verification of systems code with serval. In: Proceedings of the 27th ACM Symposium on Operating Systems Principles. p. 225–242. SOSP '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3341301.3359641
50. Nelson, L., Van Geffen, J., Torlak, E., Wang, X.: Specification and verification in the field: Applying formal methods to bpf just-in-time compilers in the linux kernel. In: Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation. OSDI'20, USENIX Association, USA (2020)
51. 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
52. Palmiotti, V.: Kernel pwning with eBPF: a love story. https://www.graplsecurity.com/post/kernel-pwning-with-ebpf-a-love-story
53. 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
54. 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
55. Shirokov, N.V.: XDP: 1.5 years in production. Evolution and lessons learned. In: Linux Plumbers Conference (2018)
56. Singh, G., Püschel, M., Vechev, M.: Fast polyhedra abstract domain. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. p. 46–59. POPL '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3009837.3009885
57. Starovoitov, A.: Bpf at facebook. https://kernel-recipes.org/en/2019/talks/bpf-at-facebook/
58. 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

59. Van Geffen, J., Nelson, L., Dillig, I., Wang, X., Torlak, E.: Synthesizing jit compilers for in-kernel dsls. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification. pp. 564–586. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_29
60. Venet, A.: Abstract cofibered domains: Application to the alias analysis of untyped programs. In: Cousot, R., Schmidt, D.A. (eds.) Static Analysis. pp. 366–382. Springer Berlin Heidelberg, Berlin, Heidelberg (1996). https://doi.org/10.1007/3-540-61739-6_53
61. 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
62. 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)
63. 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
64. Wang, X., Lazar, D., Zeldovich, N., Chlipala, A., Tatlock, Z.: Jitk: A trustworthy in-kernel interpreter infrastructure. In: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation. p. 33–47. OSDI'14, USENIX Association, USA (2014)
65. Xu, Q., Wong, M.D., Wagle, T., Narayana, S., Sivaraman, A.: Synthesizing safe and efficient kernel extensions for packet processing. In: Proceedings of the 2021 ACM SIGCOMM 2021 Conference. p. 50–64. SIGCOMM '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3452296.3472929