Verifying the Verifier: eBPF Range Analysis Verification

Harishankar Vishwanathan(✉), Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte

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

Abstract. This paper proposes an automated method to check the correctness of range analysis used in the Linux kernel's eBPF verifier. We provide the specification of soundness for range analysis performed by the eBPF verifier. We automatically generate verification conditions that encode the operation of the eBPF verifier directly from the Linux kernel's C source code and check it against our specification. When we discover instances where the eBPF verifier is unsound, we propose a method to generate an eBPF program that demonstrates the mismatch between the abstract and the concrete semantics. Our prototype automatically checks the soundness of 16 versions of the eBPF verifier in the Linux kernel versions ranging from 4.14 to 5.19. In this process, we have discovered new bugs in older versions and proved the soundness of range analysis in the latest version of the Linux kernel.

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

1 Introduction

Extended Berkeley Packet Filter (eBPF) enables the Linux kernel to be extended with user-developed functionality. Historically, eBPF has its roots in a domain-specific language for efficient packet filtering [53], wherein a user can write a description of packets that must be captured by the network stack. In its modern form, eBPF is an in-kernel register-based virtual machine with a custom 64-bit RISC instruction set. eBPF programs can be Just-in-Time (JIT) compiled to the native processor hardware with access to a subset of kernel functions and memory. Programs written in eBPF are widely used in the industry, e.g. for load balancing [10], DDoS mitigation [38], and access control [12].

eBPF Verifier. A user should be able to attach expressive programs within the operating system, while ensuring that they are safe to run. For this purpose, Linux has a built-in eBPF verifier [11] which performs a static analysis of the eBPF program to check safety properties before allowing the program

H. Vishwanathan and M. Shachnai—Equal contribution.

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13966, pp. 226–251, 2023.
https://doi.org/10.1007/978-3-031-37709-9_12

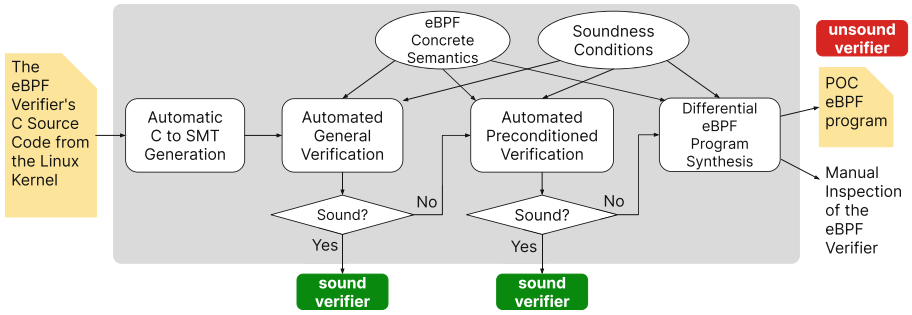

Fig. 1. Agni's methodology for automatically checking the correctness of the eBPF verifier on each commit. When we find the kernel to be unsound, we generate an eBPF program (i.e., a POC) highlighting the mismatch between abstract and concrete semantics. When we are not able to generate a POC, kernel requires a manual verification.

to be loaded. Given that the verifier is executed in a production kernel, any bug in the verifier creates a huge attack surface for exploits [50,51,62,66] and vulnerabilities [1–9,23–26,35,43–45].

Abstract Interpretation in the Kernel. The verifier, among other things, tracks the values of its variables which it subsequently uses to deem memory accesses to the kernel data structures to be safe. The eBPF static analyzer employs abstract interpretation [33] with multiple abstract domains to track the types, liveness, and values of program variables across all executions. It uses five abstract domains to track the values of variables (i.e., value tracking); four of them are variants of interval domains and the other is a bitwise domain named `tnum` [55,57,65,71]. The kernel implements abstract operators for each of these domains efficiently. Unlike traditional sound composition of sound operators typically done with abstract interpretation (i.e., modular reduced products) [31], the abstract operators are composed in a non-modular fashion. Specifically, the kernel mixes up the implementation of abstract operators in one domain with reduction operators that combine information across domains (Sect. 3, see Fig. 2(d)). Further, the Linux kernel does not provide any soundness guarantees for these operators. This makes the task of verification challenging because each abstract domain's correctness individually does not necessarily imply the correctness of their composition. To the best of our knowledge, there are no existing sound reduction operators for the abstract domains in the kernel.

This Paper. We propose an automated verification approach to check the soundness of the eBPF verifier for value tracking. To perform soundness checks on every kernel commit, we automatically generate a formula representing the actions of the abstract operator from the verifier's C code rather than manually writing them (Sect. 5). Figure 1 illustrates our workflow. We develop a general correctness specification to determine when a non-modular abstract operator that combines multiple domains is sound (Sect. 4.1). When we checked the validity of the formula generated from recent versions of the verifier with the correctness specification, we found that the verifier is unsound. We discovered that the verifier avoids man-

ifesting these soundness bugs through a shared reduction operator that preconditions the input abstract values (Sect. 4.2). Refining our correctness specification revealed that recent versions of the verifier are indeed sound.

When our refined soundness check fails, we generate a concrete eBPF program that demonstrates the mismatch between abstract values maintained by the verifier and the concrete execution of the eBPF program using program synthesis methods (Sect. 4.3). We call our approach differential synthesis because it generates programs that exercise the divergence between abstract verifier semantics and concrete eBPF semantics in unsound kernels.

Prototype and Results. We have used our prototype, Agni [18,72]., to automatically check the soundness of 16 kernel versions starting from 4.14 to 5.19. In this process, we have discovered 27 previously unknown bugs, which have been subsequently fixed by unrelated patches. For each unsound verifier, we have generated an eBPF program with at most three instructions that shows the mismatch between the semantics in $\approx 97\%$ of the cases. The eBPF programs highlighting the mismatch are smaller than previously known ones. We have also shown that the newer versions of the kernel verifier are sound with respect to value tracking. The source code for our prototype is publicly available [18,72].

2 Background on Abstract Interpretation

Abstract interpretation is a form of static analysis that uses *abstract values* from an abstract domain to represent sets of values of program variables. For example, in the interval domain, the abstract value $[x, y]$, with $x, y \in \mathbb{Z}, x \leq y$, tracks the set of concrete values $\{z \in \mathbb{Z} \mid x \leq z \leq y\}$. *Abstract operators* concisely represent the impact of the program's operations over its variables in the abstract domain.

Abstract Domains, Concretization, and Abstraction. Formally, concrete values form a partially ordered set (poset) with elements $\mathbb{C}$ and ordering relation $\sqsubseteq_{\mathbb{C}}$. The concrete poset is $\mathbb{C} \triangleq 2^{\mathbb{Z}}$ (i.e., power set of integers) with the ordering relationship $\sqsubseteq_{\mathbb{C}}$ being the subset relationship $\subseteq$. An abstract domain is also a poset, with a set of elements $\mathbb{A}$ and ordering relation $\sqsubseteq_{\mathbb{A}}$. A *concretization function* $\gamma\colon \mathbb{A} \to \mathbb{C}$, takes an abstract value $a \in \mathbb{A}$ and produces concrete values $c \in \mathbb{C}$. For example, the interval domain uses the abstract poset $\mathbb{A} \triangleq \mathbb{Z} \times \mathbb{Z}$ with the ordering relation $[x, y] \sqsubseteq_{\mathbb{A}} [a, b] \Leftrightarrow (a \leq x) \wedge (b \geq y)$.

An *abstraction function* $\alpha\colon \mathbb{C} \to \mathbb{A}$, takes a concrete value $c \in \mathbb{C}$ and produces an abstract value $a \in \mathbb{A}$. For example, in the interval domain, abstracting the concrete value $\{1, 4, 6\}$ produces $\alpha(\{1, 4, 6\}) = [1, 6]$. Concretizing $[1, 6]$ yields $\gamma([1, 6]) = \{1, 2, 3, 4, 5, 6\}$. As seen in this example, the abstraction of a concrete value may over-approximate it to maintain concise representation in the abstract domain. A value $a \in \mathbb{A}$ is a *sound abstraction* of $c \in \mathbb{C}$ if $c \sqsubseteq_{\mathbb{C}} \gamma(a)$. For a sound abstraction a of c, the smaller the concrete value $\gamma(a)$, the higher the *precision* of the abstraction.

Abstract Operators. Intuitively, abstract operators capture the computation of concrete operators over program variables in the abstract domain. For example, in the range domain, the action of concrete unary negation $-_{\mathbb{C}}(\cdot)$ may be

abstracted by $-_{\mathbb{A}}([x,y]) \triangleq [-y,-x]$. Consider a concrete operation $f: \mathbb{Z}_n \to \mathbb{Z}_n$ on a single program variable that is an n-bit value. We can lift f point-wise to any set $c \in \mathbb{C}$, where $f(c) \triangleq \{f(z) \mid z \in c\}$. An abstract operator $g: \mathbb{A} \to \mathbb{A}$ is a *sound abstraction* of f if $\forall a \in \mathbb{A} : f(\gamma(a)) \sqsubseteq_{\mathbb{C}} \gamma(g(a))$.

Galois Connection. Abstraction and concretization functions (α, γ) are said to form a Galois connection if: (1) α is monotonic (i.e. $x \sqsubseteq_{\mathbb{C}} y \implies \alpha(x) \sqsubseteq_{\mathbb{A}} \alpha(y)$), (2) γ is monotonic ($a \sqsubseteq_{\mathbb{A}} b \implies \gamma(a) \sqsubseteq_{\mathbb{C}} \gamma(b)$), (3) $\gamma \circ \alpha$ is extensive (i.e. $\forall c \in \mathbb{C} : c \sqsubseteq_{\mathbb{C}} \gamma(\alpha(c))$), and (4) $\alpha \circ \gamma$ is reductive (i.e. $\forall a \in \mathbb{A} : \alpha(\gamma(a)) \sqsubseteq_{\mathbb{A}} a$) [56].

The Galois connection is denoted as $(\mathbb{C}, \sqsubseteq_{\mathbb{C}}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathbb{A}, \sqsubseteq_{\mathbb{A}})$. The existence of a Galois connection enables reasoning about the soundness and the precision of any abstract operator. It is in principle possible to compute a sound and precise abstraction of any concrete operator f through the composition $\alpha \circ f \circ \gamma$. However, it is computationally expensive, due to the evaluation of the concretization γ.

Combining Multiple Abstract Domains Through Cartesian Product [31]. Suppose we are given two abstract domains (sets $\mathbb{A}_1, \mathbb{A}_2$) with sound abstraction functions $\alpha_{\mathbb{A}1}, \alpha_{\mathbb{A}2}$ and concretization functions $\gamma_{\mathbb{A}1}, \gamma_{\mathbb{A}2}$. The Cartesian product abstract domain uses the set $\mathbb{P} \triangleq \mathbb{A}_1 \times \mathbb{A}_2$, and the ordering relationship applied separately to each domain: $(a_1 \sqsubseteq_{\mathbb{A}1} b_1) \wedge (a_2 \sqsubseteq_{\mathbb{A}2} b_2) \Rightarrow (a_1, a_2) \sqsubseteq_{\mathbb{P}} (b_1, b_2)$. The concretization function intersects the results obtained from concretizing each element in its respective abstract domain: $\gamma_{\mathbb{P}}(a_1, a_2) \triangleq \gamma_{\mathbb{A}1}(a_1) \cap \gamma_{\mathbb{A}2}(a_2)$. For a concrete value $c \in \mathbb{C}$, the abstraction functions are applied domain-wise and combined: $\alpha_{\mathbb{P}}(c) \triangleq \big(\alpha_{\mathbb{A}1}(c), \alpha_{\mathbb{A}2}(c)\big)$. The Cartesian product domain enjoys a Galois connection $(\mathbb{C}, \sqsubseteq_{\mathbb{C}}) \underset{\alpha_{\mathbb{P}}}{\overset{\gamma_{\mathbb{P}}}{\leftrightarrows}} (\mathbb{P}, \sqsubseteq_{\mathbb{P}})$ building on the Galois connections of its component abstract domains.

For example, consider the interval domain ($\mathbb{A}_1, \sqsubseteq_{\mathbb{A}1}$ defined as above) and the parity domain ($\mathbb{A}_2 \triangleq \{\bot, \mathit{odd}, \mathit{even}, \top\}$ with ordering relationships $\bot \sqsubseteq_{\mathbb{A}2} \mathit{odd}, \mathit{even} \sqsubseteq_{\mathbb{A}2} \top$). Suppose at some point the two interpretations produce abstract values $[3,5]$ and *even* in the two domains. The concretization of the Cartesian product abstract value $([3,5], \mathit{even})$ produces the set $\{4\}$, which is smaller than the concretizations of either abstract value $[3,5]$ or *even* in their respective domains. However, since the abstraction functions are applied domain-wise, such information cannot be propagated to the abstract values themselves. For example, it is desirable to propagate information from the abstract value *even* in $\mathbb{A}_2$ to reduce the interval to $[4,4]$ in $\mathbb{A}_1$.

Reduced Products. Intuitively, we wish to make an abstract value in one domain more precise using information available in an abstract value in a different domain. Suppose we are given an abstract value (a_1, a_2) from the Cartesian product domain. A *reduction operator* [34] attempts to find the smallest abstract value (a_1', a_2') such that its concretization is the same as that of (a_1, a_2), i.e. $\gamma_{\mathbb{A}1}(a_1) \cap \gamma_{\mathbb{A}2}(a_2)$. Formally, the reduction operator $\rho: \mathbb{P} \to \mathbb{P}$ is defined as the greatest lower bound of all abstract values whose concretization is larger than that of the given abstract value,

i.e. $\rho(a_1, a_2) \triangleq \bigsqcap_{\mathbb{P}} \{(a_1', a_2') \mid \gamma_{\mathbb{P}}(a_1, a_2) \sqsubseteq_{\mathbb{C}} \gamma_{\mathbb{P}}(a_1', a_2')\}$.

However, this definition is impractical to compute even on finite domains.

In general, more "relaxed" versions of reduction operators may be designed to improve precision with efficient computation. For example, Granger [40] introduces a set of reduction operators ρ_1, ρ_2 to reduce each abstract domain in turn, using information from the other, until a fixed point. The operator $\rho_1 : \mathbb{A}_1 \times \mathbb{A}_2 \to \mathbb{A}_1$ reduces the abstract value in domain $\mathbb{A}_1$, while $\rho_2 : \mathbb{A}_1 \times \mathbb{A}_2 \to \mathbb{A}_2$ reduces that in $\mathbb{A}_2$. The reduction using ρ_1 is sound if $\forall a_1 \in \mathbb{A}_1, a_2 \in \mathbb{A}_2 : \gamma_{\mathbb{P}}(\rho_1(a_1, a_2), a_2) = \gamma_{\mathbb{P}}(a_1, a_2)$ (preserve concrete values in the intersection) and $\rho_1(a_1, a_2) \sqsubseteq_{\mathbb{A}1} a_1$ (improve precision). Similarly, reduction using ρ_2 is sound if $\forall a_1 \in \mathbb{A}_1, a_2 \in \mathbb{A}_2 : \gamma_{\mathbb{P}}(a_1, \rho_2(a_1, a_2)) = \gamma_{\mathbb{P}}(a_1, a_2)$ and $\rho_2(a_1, a_2) \sqsubseteq_{\mathbb{A}2} a_2$.

3 Abstract Interpretation in the Linux Kernel

The Linux kernel implements abstract interpretation to check the safety of eBPF programs loaded into the kernel. The kernel's algorithms are encoded into a component called the *eBPF verifier,* which is a part of the pre-compiled operating system image. The Linux kernel uses several abstract domains to track the type, liveness, and values of registers and memory locations used by eBPF programs. Among these, the abstract domains used by the kernel to track values are critical since they are used to guard statically against malicious programs that may access kernel memory. In Linux kernel v5.19 (latest as of this writing), these analyses constitute roughly 2100 lines of source code in the eBPF verifier. Implementing such analyses soundly in the kernel is challenging. This part of the verifier has been a source of several high-profile security vulnerabilities [1–9,23–26,35,43–45] and exploits [50,51,62,66].

The Linux kernel uses five abstract domains for value tracking, including intervals in unsigned 64-bit (`u64`), unsigned 32-bit (`u32`), signed 64-bit (`s64`), signed 32-bit (`s32`), and tri-state numbers (`tnum` [61,71]). The kernel does not provide a formal specification of their abstraction or concretization functions, or proofs of soundness of the abstract operators. Below, we illustrate the abstract domains used in the Linux kernel with the unsigned 64-bit interval domain `u64` and tristate numbers `tnum`.

The `u64` Domain. The `u64` abstract domain tracks an upper and lower bound of a 64-bit register interpreted as an unsigned 64-bit value. The eBPF verifier maintains the abstract `u64` value as part of its static state for each register. Figure 2(a) provides a simplified C source code for abstract addition in the `u64` domain. The operator takes two abstract values `in1` and `in2`, with the two components of each abstract value denoted by the members `u64_min` and `u64_max`. The output abstract value is stored in `out`. Here, `U64_MAX` is the largest 64-bit non-negative integer. The first `if` condition detects if integer overflows may occur as a result of addition. If there is overflow, the analysis loses all precision, setting the 64-bit bounds of the result to the largest abstract value, [0, `U64_MAX`]. If there is no overflow (`else` clause), `out` is set to the component-wise sum of the bounds of `in1` and `in2`, similar to unbounded bit-width interval arithmetic [32].

Formally, the abstract domain 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 ordering relationship is $(x_1 \geq_{u64}$

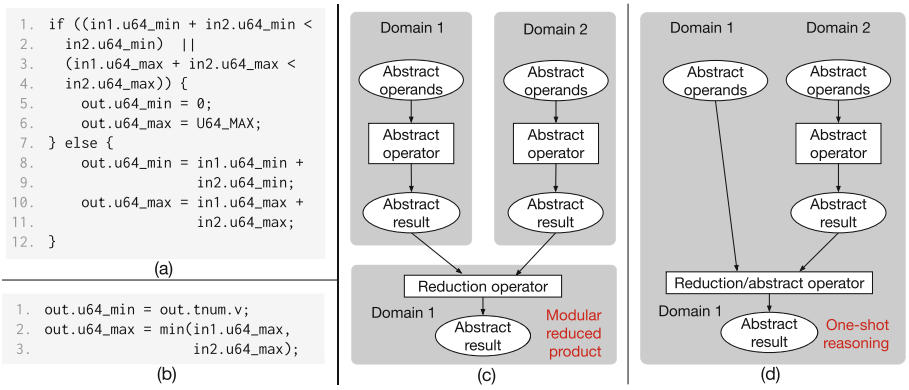

Fig. 2. Excerpts (simplified) from the kernel's implementation of the abstract operators for (a) addition (from the function `scalar_min_max_add` [14]), and (b) bitwise AND (from `scalar_min_max_and` [15]). (c) Example of reduced product abstract interpretation where one may use inductive assertions on abstract operators from each domain, along with the soundness of reduction operators, to reason about the correctness of the overall abstraction. The greyed boxes show modular reasoning about components within the boxes. (d) In the Linux kernel, it is challenging to reason modularly about the correctness of abstract operators in each domain independently from their pairwise reductions, since the implementation combines abstraction with reduction. Proving soundness requires one-shot reasoning about all operations together.

$x_2) \wedge (y_1 \leq_{u64} y_2) \Leftrightarrow [x_1, y_1] \sqsubseteq_{u64} [x_2, y_2]$. 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.

Tristate Numbers (`tnums`). This abstract domain in the Linux kernel tracks which bits of a variable are known to be 0, known to be 1, or unknown across executions of the program. This domain is similar to bitwise domains [55,57,65]. However, the kernel implements this abstract domain efficiently with a tuple of two unsigned integers (`v`,`m`). If `m` for a particular bit is 1, then the value of that bit is unknown. If `m` for a particular bit is 0, then value of that bit is equal to `v`'s value for the particular bit. More formally, the abstraction function (α_t) is written using two other functions defined as follows: $\alpha_{\&}(C) \triangleq \&\{c \mid c \in C\}$; and $\alpha_{|}(C) \triangleq |\{c \mid c \in C\}$. Then, $\alpha_t(C) \triangleq (\alpha_{\&}(C), \alpha_{\&}(C) \hat{} \alpha_{|}(C))$. The concretization function is written as: $\gamma_t(P) = \gamma_t((P.\mathtt{v}, P.\mathtt{m})) \triangleq \{c \in \mathbb{Z}_{64}^{+} \mid c \,\&\, P.\mathtt{m} = P.\mathtt{v}\}$ [71].

Abstract Operators In The Linux Kernel and Challenges in Proving their Correctness. The Linux kernel implements an abstract operator in each abstract domain for each arithmetic and logic (ALU) instruction and each jump instruction in the eBPF instruction set.[1] The kernel verifier also provides

[1] The ALU instructions include 32 and 64-bit `add`, `sub`, `mul`, `div`, `or`, `and`, `lsh`, `rsh`, `neg`, `mod`, `xor`, `arsh` and the jump instructions include 32 and 64-bit `ja`, `jeq`, `jgt`, `jge`, `jlt`, `jle`, `jset`, `jne`, `jsgt`, `jsge`, `jslt`, `jsle` [13].

functions to propagate information between the abstractions (reductions). However, it does not provide formal underpinnings, e.g. Galois connections. The overall analysis appears to be a Reduced Product abstract interpretation (Sect. 2).

However, the key challenge in proving soundness is that the kernel's operators combine abstraction with reduction. Consider the excerpt in Fig. 2(b) from the implementation of the bitwise AND operation in the `u64` abstract domain in the kernel, simplified for clarity. As before, `in1` and `in2` correspond to the input abstract values, and `out` to the output abstract value. The members with names `tnum.*` denote the components of the abstract tnum. Before the execution of these two lines, the `tnum` abstract output `out.tnum.v` has already been computed. In the first line, the lower bound of the `u64` result, `out.u64_min` is updated using the output abstract value in a different domain (`out.tnum.v`). Hence, the operation overall is not (merely) an abstract operator in the `u64` domain. In the second line, the output abstract state `out.u64_max` is updated using the abstract *inputs* in the `u64` domain. Reduction operators consume abstract outputs, not inputs. Hence, the operation overall is not a reduction operator either.

These characteristics apply not just to the kernel's bitwise AND operation in the `u64` domain. Figure 2(d) shows the structure of several of the kernel's abstract operators, compared against the typical structure of product domains and reduction operators (Fig. 2(c)). The kernel's algorithms combine abstraction with reduction, making it challenging to prove their soundness in a modular fashion. Instead, we must resort to a "one-shot" approach, which attempts to prove the soundness of the abstraction of an operator in one domain and the reductions across domains together. We call the kernel's abstract operators *abstraction/reduction operators* in the rest of this paper.

4 Automatic Verification of the Kernel's Algorithms

Given the non-modular structure of the kernel's abstract algorithms (Sect. 3), we cannot use traditional methods to prove their soundness, i.e. by showing the soundness of each domain and the reductions separately. Further, the kernel's algorithms have been evolving continuously with the inclusion of new features to the eBPF run-time environment. We want our methods to be applicable to every new update and commit to the Linux kernel.

Hence, our goal is to perform automatic verification using SMT solvers to prove the soundness of (or find bugs in) the C implementation of Linux's abstraction/reduction operators. We work with the input-output semantics of the kernel's abstraction/reduction operators in first-order logic extracted automatically from the kernel's C source code (details of the extraction deferred to Sect. 5).

Overview of Our Approach. We develop generic soundness specifications for the Linux kernel's abstraction/reduction operators, handling arithmetic, logic, and branching instructions (Sect. 4.1). We find that several kernel operators violate these soundness specifications. However, many of these violations flag latent bugs in the kernel's algorithms—bugs which are not necessarily manifested in concrete program executions. We observe that the kernel includes a shared "tail" of computation in all of its abstraction/reduction operators. We use this shared compu-

tation to refine our soundness specification by preconditioning the input abstract states (Sect. 4.2). This refinement enables proving the soundness of several of the kernel's operators. However, it still identifies many potential violations of soundness in the kernel. We present a method based on program synthesis to generate loop-free eBPF programs that manifest the bugs identified by the soundness specifications, automatically producing programs that have divergent concrete and abstract semantics. We call this method differential synthesis (Sect. 4.3).

Figure 1 illustrates our entire workflow. Starting from the Linux kernel source code, our techniques produce concrete eBPF programs that manifest soundness bugs in the kernel's algorithms. We have used this procedure to prove the soundness of multiple Linux kernel versions, discovered previously unknown soundness bugs (i.e. no CVEs assigned, to our knowledge), with validated proof-of-concept programs triggering those bugs.

4.1 Soundness Specification for Abstraction/Reduction Operators

We present verification conditions that are *sufficient* to assert the soundness of abstraction/reduction operators in the Linux kernel.

Preliminaries. Encoding Soundness for a Single Abstract Domain in SMT. We describe how to encode the soundness condition for an abstract operator of two operands as an SMT formula, since most eBPF instructions take two operands. Suppose $f: \mathbb{C} \times \mathbb{C} \to \mathbb{C}$ is a binary concrete operation (e.g. 64-bit addition) over the concrete domain (e.g. $\mathbb{C} \triangleq 2^{\mathbb{Z}^+_{64}}$). Suppose the operator $g: \mathbb{A} \times \mathbb{A} \to \mathbb{A}$ abstracts f. Operator g is sound (Sect. 2) 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 can check soundness with an SMT query as follows. Suppose we have SMT variables to denote a bitvector $x \in \mathbb{C}$ and an abstract value $a \in \mathbb{A}$. We can use the concretization function γ to represent the fact that x is included in the concretization of a. For example, for the `u64` domain, we may use the formula $mem_{u64}(x, a) \triangleq (a.min \leq_{u64} x) \wedge (x \leq_{u64} a.max)$ to assert that $x \in \gamma(a)$.

The input-output relationship of abstract operator g is available as a first-order logic formula extracted from the kernel source code (Sect. 5). We represent the resulting formula 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 concrete semantics of the eBPF instruction set determines the input-output relationship of the concrete operation f. For example, the `bpf_add64` instruction performs binary addition (with possibility of overflow) of two 64-bit registers, denoted by $+_{64}$. The action of this instruction is encoded through the formula $x^o = conc_f(x^i_1, x^i_2)$; for `bpf_add64`, $conc_f(x^i_1, x^i_2) \triangleq (x^i_1 +_{64} x^i_2)$.

The concrete ordering relationship $\sqsubseteq_{\mathbb{C}}$ is just the subset operation $\subseteq$ between two sets. For two sets S_1, S_2, we can encode the relationship $S_1 \subseteq S_2$ by asserting that $\forall x : x \in S_1 \Rightarrow x \in S_2$. Putting all this together, we can check the soundness of a single abstract operator abs_g, by using an SMT solver to check the validity of the formula (i.e., by checking if the negation is unsatisfiable).

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

Generalizing Soundness To Abstraction/Reduction Operators Spanning Multiple Abstract Domains. For the abstraction/reduction operators in Linux (Sect. 3), we can no longer assert soundness for an abstract domain purely using abstract values from that domain. We show how to extend the reasoning to two abstract domains. Let us denote the two abstract domains by $\mathbb{A}_1$ and $\mathbb{A}_2$. An eBPF instruction has two inputs (x_1^i, x_2^i) and each input has the corresponding abstract value for each abstract domain. Suppose a_{11}^i and a_{12}^i correspond to abstract values for the first input from domains $\mathbb{A}_1$ and $\mathbb{A}_2$, respectively (similarly, a_{21}^i and a_{22}^i for the second input). Further, the concrete input x^i must be in the intersection of the concretizations of all its abstract values. Hence, the formula $mem_{\mathbb{A}_1}(x_1^i, a_{11}^i) \wedge mem_{\mathbb{A}_2}(x_1^i, a_{12}^i) \wedge mem_{\mathbb{A}_1}(x_2^i, a_{21}^i) \wedge mem_{\mathbb{A}_2}(x_2^i, a_{22}^i)$ must hold.

We denote the kernel's abstraction/reduction operation, extracted from C source code, as $\{a_1^o, a_2^o\} = abs_g(a_{11}^i, a_{12}^i, a_{21}^i, a_{22}^i)$. Note that the kernel's operation outputs a list of abstract values corresponding to each abstract domain (unlike Eq. 1). The concrete semantics dictates that $x^o = conc_f(x_1^i, x_2^i)$.

To establish the soundness of the abstraction/reduction operator, we ensure that the concrete output is included in the concretizations of the abstract outputs in each domain, i.e., $mem_{\mathbb{A}_1}(x^o, a_1^o) \wedge mem_{\mathbb{A}_2}(x^o, a_2^o)$. Putting it all together, we check the validity of the following SMT formula:

$$\begin{aligned} \forall x_1^i,\ \ x_2^i \in \mathbb{C},\ \ a_{11}^i,\ \ a_{21}^i \in \mathbb{A}_1,\ a_{12}^i,\ a_{22}^i \in \mathbb{A}_2 : \\ mem_{\mathbb{A}_1}(x_1^i, a_{11}^i) \wedge mem_{\mathbb{A}_2}(x_1^i, a_{12}^i) \wedge mem_{\mathbb{A}_1}(x_2^i, a_{21}^i) \wedge mem_{\mathbb{A}_2}(x_2^i, a_{22}^i) \wedge \\ x^o = conc_f(x_1^i, x_2^i) \wedge \{a_1^o, a_2^o\} = abs_g(a_{11}^i, a_{12}^i, a_{21}^i, a_{22}^i) \\ \Rightarrow (mem_{\mathbb{A}_1}(x^o, a_1^o) \wedge mem_{\mathbb{A}_2}(x^o, a_2^o)) \end{aligned} \tag{2}$$

The kernel uses five abstract domains (Sect. 3). Extending from two domains to all five domains is straightforward. It involves the addition of membership queries for the inputs and the corresponding abstract values (i.e., *mem* predicate above). The encoding of each of the kernel's abstraction/reduction operators returns a list containing five abstract outputs (one for each domain). Finally, we check that the concrete output is included in the concretization of each abstract output.

Encoding Arithmetic and Logic (ALU) Instructions. Using the formulation above, we have encoded soundness specifications of abstraction/reduction operators for 16 eBPF ALU instructions, which include 32 and 64-bit `add`, `sub`, `div`, `or`, `and`, `lsh`, `rsh`, `neg`, `mod`, `xor`, `arsh`. Notably, we exclude the multiplication instruction `mul`, whose SMT formula involves a bitvector multiplication operation and a large unrolled loop, making it intractable in the bitvector theory.

Encoding Branch Instructions. We also encoded soundness specifications for conditional and unconditional branches (`jeq`, `jlt`, etc.) on both 64 and 32-bit register operands. These amount to 20 instructions, for a total of 36 instructions captured by our encodings. While the soundness of abstracting ALU instructions follows the general structure of Eq. 2, writing down the soundness conditions for branches is more involved. Branches do not concretely modify their input registers. However, the kernel learns new information in the abstract domains using the branch outcome (true vs. false). For example, in the `u64` domain, consider

two abstract registers $[1,5],[3,3]$. Jumping upon an = (equals) comparison shows that the first register can also be set to $[3,3]$ in the true case. Indeed, each conditional jump instruction produces *four* abstract outputs (rather than the usual *one* output for ALU instructions), corresponding to updated abstract values for two registers across two branch outcomes.

We illustrate the encoding of the correctness condition for the jump instruction for a single abstract domain. Given two concrete operands x_1^i and x_2^i, the concrete interpretation for the jump instruction returns whether the condition is true or false. When $x^o = conc_f(x_1^i, x_2^i)$, x^o will be either true or false. The kernel's abstraction/reduction operator generates four output abstract values, $a_{1t}^o, a_{1f}^o, a_{2t}^o, a_{2f}^o$. There are two abstract outputs corresponding to each input. They reflect the updated abstract value for the true case (e.g., a_{1t}^o is the updated abstract value of the first input when the branch condition is true), and similarly for the false case. We represent the kernel's abstraction/reduction operator for branch instructions by the formula $\{a_{1t}^o, a_{1f}^o, a_{2t}^o, a_{2f}^o\} = abs_g(a_1^i, a_2^i)$.

Our correctness condition for jumps requires that the inputs are present in the concretizations of the corresponding abstract value in both the true and false branch outcomes. The formula below specifies this correctness condition.

$$\begin{aligned} \forall x_1^i, \quad x_2^i \in \mathbb{C}, \quad a_1^i, \quad a_2^i \in \mathbb{A} : mem_{\mathbb{A}}(x_1^i, a_1^i) \wedge mem_{\mathbb{A}}(x_2^i, a_2^i) \wedge \\ x^o = conc_f(x_1^i, x_2^i) \wedge \{a_{1t}^o, a_{1f}^o, a_{2t}^o, a_{2f}^o\} = abs_g(a_1^i, a_2^i) \Rightarrow \\ ((x^o \Rightarrow (mem_{\mathbb{A}}(x_1^i, a_{1t}^o) \wedge mem_{\mathbb{A}}(x_2^i, a_{2t}^o))) \wedge \\ (\neg x^o \Rightarrow (mem_{\mathbb{A}}(x_1^i, a_{1f}^o) \wedge mem_{\mathbb{A}}(x_2^i, a_{2f}^o)))) \end{aligned} \tag{3}$$

The above correctness condition can be extended to multiple domains in a manner similar to Eq. 2. The kernel's implementation of the abstraction/reduction operator for a single jump instruction produces 20 output abstract values (2 inputs $\times$ 2 branch outcomes $\times$ 5 domains).

4.2 Refining Soundness Specification with Input Preconditioning

When we checked the soundness of the kernel's verifier using the soundness specifications in Sect. 4.1, we observed that many of the abstract operators are not sound. However, it is unclear whether these violations are latent unsound behaviors, or behaviors that could actually manifest with concrete eBPF programs. Specifically, the precondition in Eq. 2 is too general, including any combination of abstract values (across domains) as long as the intersection of their concretizations is non-empty. Indeed, the abstract operators in the Linux kernel are unsound if each instruction may start from any arbitrary abstract value across domains. However, these combinations of abstract values may never be encountered in any eBPF program. Our goal is to refine the soundness specifications from Sect. 4.1 to minimize reporting latent (but unmanifested) bugs.

Shared Suffix of Abstraction/Reduction Operator. Upon carefully analyzing the kernel's abstraction/reduction operators, we observed that the kernel performs certain common computations—a *shared suffix* of abstraction/reduction operations—right before producing each abstract output (Fig. 3(a)). As a concrete example, in kernel version 5.19, the function

`reg_bounds_sync` is called at the end of each ALU operation [49], updating the signed domains using the unsigned domains, the `u64` bounds from `u32` bounds and `tnums`, besides other reductions [48].

Our key insight is that this shared suffix of abstraction/reduction has the effect of preconditioning the initial abstract values for any subsequent instruction, narrowing down the set of possible abstract values that a subsequent instruction may encounter as input. Further, all eBPF programs start executing from abstract values where each register in every domain is either $\top$ (any concrete value in the domain) or its concretization is a singleton (precisely known concrete value). We observe and show using an SMT solver that the shared suffix computation does not modify initial values.

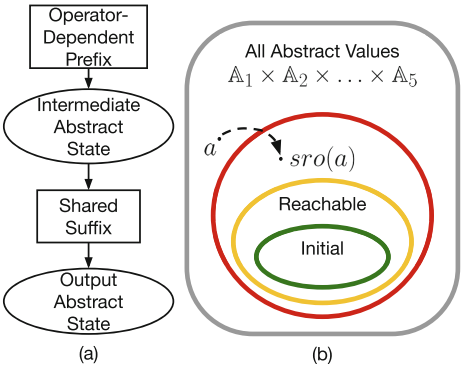

Fig. 3. (a) The structure of each abstraction/reduction operator in the kernel can be conceptualized as having a prefix that depends on the specific operator, generating an intermediate output, and a suffix that is shared across all the operators, resulting in the final abstract output. (b) We use a refined soundness specification that preconditions input abstract values a using the shared suffix $sro(.)$ of the reduction operators used in the Linux kernel.

Refined Soundness Specification by Preconditioning Input Abstract Values. We can leverage shared suffix operations to refine our soundness specification as follows. First, let $sro(a)$ denote the abstract outputs of computing the shared suffix of the abstraction/reduction over the abstract inputs $a \in \mathbb{A}_1 \times \mathbb{A}_2 \cdots \times \mathbb{A}_5$. The SMT formula encoding $sro(a)$ is extracted using our C to SMT encoder (Sect. 5). The main change from the specifications in Sect. 4.1 is that the shared suffix preconditions the input values to any abstract operator. Hence, for example, the soundness specification for two abstract domains from Eq. 2 is updated to use an input abstract value $sro(a)$ as shown below:

$$\begin{aligned} \forall x_1^i,\ \ x_2^i \in \mathbb{C},\ \ a_{11}^i,\ \ a_{21}^i \in \mathbb{A}_1,\ \ a_{12}^i,\ \ a_{22}^i \in \mathbb{A}_2 : \\ (b_{11}^i, b_{12}^i) = sro(a_{11}^i, a_{12}^i) \wedge (b_{21}^i, b_{22}^i) = sro(a_{21}^i, a_{22}^i) \wedge \\ mem_{\mathbb{A}_1}(x_1^i, b_{11}^i) \wedge mem_{\mathbb{A}_2}(x_1^i, b_{12}^i) \wedge mem_{\mathbb{A}_1}(x_2^i, b_{21}^i) \wedge mem_{\mathbb{A}_2}(x_2^i, b_{22}^i) \wedge \\ x^o = conc_f(x_1^i, x_2^i) \wedge \{a_1^o, a_2^o\} = abs_g(b_{11}^i, b_{12}^i, b_{21}^i, b_{22}^i) \\ \Rightarrow (mem_{\mathbb{A}_1}(x^o, a_1^o) \wedge mem_{\mathbb{A}_2}(x^o, a_2^o)) \end{aligned} \tag{4}$$

It is straightforward to generalize to multiple domains. Refinement eliminated most of the latent violations reported from Sect. 4.1. We found that the latest kernel versions are sound with respect to value tracking.

4.3 Automatically Producing Programs Exercising Soundness Bugs

Even after refining the soundness specifications (Sect. 4.2), we still find a few violations of soundness. It is challenging to determine whether these violations are "real" (manifested in actual eBPF programs) or latent, since input abstract values preconditioned by *sro* still overapproximate the abstract values that may occur when analyzing actual eBPF programs (Fig. 3(b), Sect. 4.2).

We aim to automatically generate eBPF programs that manifest soundness bugs (uncovered by the techniques in Sect. 4.2) in an actual kernel verifier execution. Our problem is a form of *differential synthesis*: generating programs whose semantics diverge between the concrete execution and the abstract analysis. We propose a sound but incomplete approach to generate eBPF programs that demonstrate soundness violations. We enumerate loop-free programs up to a bounded length, using an SMT solver to identify concrete and abstract operands that manifest soundness violations.

Our approach is a combination of well-known existing techniques from enumerative [20,52,63] and deductive program synthesis [19,41,58,67]. However, unlike typical program synthesis problems which have a $\forall\exists$ formula structure (e.g. meet a specification on all inputs), our problem has a much more tractable $\exists$ structure, i.e. finding one concrete input and program to trigger a soundness violation. In this sense, it is more akin to property-directed reachability algorithms used in model checking [22,27].

Preliminaries. The eBPF run-time starts executing eBPF programs with all live registers holding values that are either precisely known at compile time (e.g. offsets into valid memory regions) or completely unknown (e.g. contents of packet memory). For an abstract value $a \in \mathbb{A}_1 \times \mathbb{A}_2 \cdots \times \mathbb{A}_5$, we say that $init(a)$ holds if a is either singleton (e.g. $\forall x \in \mathbb{Z}_{64}^{+} : [x, x]$ in `u64`) or $\top$ in each domain $\mathbb{A}_i$. We refer to such abstract values as *initial abstract values*. It is straightforward to write down an SMT formula for $init(a)$ for the kernel's domains. We say an abstract value $b \in \mathbb{A}_1 \times \mathbb{A}_2 \cdots \times \mathbb{A}_5$ is *reachable* if there exists a sequence of eBPF instructions for which the abstract analysis can produce b for some register starting from input registers whose abstract values all satisfy $init(\cdot)$.

Overview. Given an abstract operator that violates the soundness specification in Sect. 4.2, our algorithm finds an eBPF instruction sequence that shows that the violating input abstract values are reachable. For a bounded program length k, we enumerate all sequences of eBPF concrete operators (i.e. arithmetic, logic, and branching instructions) of length $k-1$, with the k^{th} instruction being the violating concrete operator. This enumeration produces the "skeleton" of the program, filling out the opcodes, but leaving the operands as well as the data and control flow undetermined. For each skeleton, we discharge an SMT query that identifies the concrete and abstract operands for k instructions with well-formed data and control flow. The first instruction consumes eBPF initial abstract values. Starting from $k = 1$, if we cannot find an eBPF program of length k that manifests the violation, we increment k and try again until a timeout.

Single Instruction Programs ($k = 1$). As the base case, we check whether initial abstract values along with suitable concrete values may already violate

soundness (Sect. 4.2). For example, suppose our enumeration generated the 1-instruction program $v = \texttt{bpf\_or}(t, u)$. For simplicity, below we work with just one abstract domain. Building on Eq. (1), we discharge the SMT formula:

$$\begin{aligned} t, u \in \mathbb{C}, \quad a_t, a_u \in \mathbb{A} : \\ init(a_t) \wedge init(a_u) \wedge mem_{\mathbb{A}}(t, a_t) \wedge mem_{\mathbb{A}}(u, a_u) \ \wedge \\ v = conc_{or}(t, u) \wedge a_v = abs_{or}(a_t, a_u) \wedge \neg(mem_{\mathbb{A}}(v, a_v)) \end{aligned} \tag{5}$$

If the formula is satisfiable, the model provides the concrete operands t, u, with the result that $\texttt{bpf\_or}(t, u)$ is an executable eBPF program manifesting the soundness violation. However, an unsound operator may fail to produce a model since the necessary abstract operands lie outside the initial abstract values.

Straight-line Programs, Length $k > 1$. Larger the length of the program k, larger the set of reachable input abstract values available to manifest a soundness violation at the k^{th} instruction. We exhaustively enumerate all possible $(k-1)$-long instruction sequences. To enable well-formed data flow between the k instructions, the inputs for each instruction are sourced either from the outputs of prior instructions or initial abstract values.

For example, consider a two-instruction program ($k = 2$) generated by the enumerator: `r = bpf_and(p,q); v = bpf_or(t,u)`, We are looking for soundness violation in `bpf_or`. The variables p, q, r, t, u, v are concrete values, with corresponding abstract values $a_p, a_q, \cdots, a_v$. The abstract inputs of the first instruction `bpf_and` are initial abstract values. The abstract inputs of the last instruction may be drawn from either a_p, a_q, a_r or the initial abstract values. We use the formula $assign(x, \{y_1, y_2, \cdots\})$ to denote that x is mapped to one of the variables $y_1, y_2, \cdots$ in both the concrete and abstract domains. We can write down $assign(x, \{y_1, y_2, \cdots\}) \triangleq (x = y_1 \wedge a_x = a_{y_1}) \vee (x = y_2 \wedge a_x = a_{y_2}) \vee \cdots$. We discharge the following SMT formula to a solver:

$$\begin{aligned} p, q, r, t, u, v \in \mathbb{C}, \quad a_p, a_q, a_r, a_t, a_u, a_v \in \mathbb{A} : \\ init(a_p) \wedge init(a_q) \wedge mem_{\mathbb{A}}(p, a_p) \wedge mem_{\mathbb{A}}(q, a_q) \ \wedge \\ r = conc_{and}(p, q) \wedge a_r = abs_{and}(a_p, a_q) \wedge mem_{\mathbb{A}}(r, a_r) \wedge \\ (init(a_t) \vee assign(t, \{p, q, r\})) \wedge (init(a_u) \vee assign(u, \{p, q, r\})) \wedge \\ mem_{\mathbb{A}}(t, a_t) \wedge mem_{\mathbb{A}}(u, a_u) \wedge \\ v = conc_{or}(t, u) \wedge a_v = abs_{or}(a_t, a_u) \wedge \neg(mem_{\mathbb{A}}(v, a_v)) \end{aligned} \tag{6}$$

A model for the formula produces the concrete and abstract operands for the two instructions, leading to an executable bug-manifesting program. This approach is extensible to more instructions and more abstract domains.

Loop-free Programs. Incorporating branch instructions significantly broadens the set of input abstract values available to the k^{th} instruction, improving the likelihood of finding a bug-manifesting program at a given length. We turn each branch into a single-instruction `ite` whose outputs are available for subsequent instructions. More concretely, (i) any of the $1 \cdots k-1$ instructions may be `jump` instructions; (ii) the jump target of a branch instruction in the i^{th} slot for both outcomes (i.e. true or false) points to the $i + 1^{th}$ slot, and (iii) the abstract

outputs of the branch (e.g. from Eq. (3)) may be used as abstract inputs for subsequent instructions, similar to arithmetic and logic instructions.

As an example, suppose our enumerator produces `r = bpf_jump_gt64(p,q,0); v = bpf_or(t,u)`. Here r is a concrete value which is either $true$ or $false$. We use 0 as the jump target, always pointing branches to the next instruction. There are four abstract outputs from the jump: a_{pt}, a_{qt} for the true branch and a_{pf}, a_{qf} for the false branch (see Sect. 4.1). For convenience, we set the abstract value a_p^o (resp. a_q^o) to either a_{pt} or a_{pf} (resp. a_{qt} or a_{qf}) based on the branch outcome; and also assert that the corresponding final concrete values $p^o = p$ and $q^o = q$. Building on Eq. (3), we ask the SMT solver for a model of the formula:

$$
\begin{aligned}
p, q, t, u, v \in \mathbb{C}, \quad r \in \{true, false\}, \quad a_p, a_q, a_t, a_u, a_v \in \mathbb{A} : \\
init(a_p) \wedge init(a_q) \wedge mem_{\mathbb{A}}(p, a_p) \wedge mem_{\mathbb{A}}(q, a_q) \wedge \\
r = conc_{jump\_gt64}(p, q) \wedge \{a_{pt}, a_{pf}, a_{qt}, a_{qf}\} = abs_{jump\_gt64}(a_p, a_q) \wedge \\
(r \Rightarrow (mem_{\mathbb{A}}(p, a_{pt}) \wedge mem_{\mathbb{A}}(q, a_{qt}) \wedge a_p^o = a_{pt} \wedge a_q^o = a_{qt})) \wedge \\
(\neg r \Rightarrow (mem_{\mathbb{A}}(p, a_{pf}) \wedge mem_{\mathbb{A}}(q, a_{qf}) \wedge a_p^o = a_{pf} \wedge a_q^o = a_{qf})) \wedge \\
(init(a_t) \vee assign(t, \{p^o, q^o\})) \wedge (init(a_u) \vee assign(u, \{p^o, q^o\})) \wedge \\
mem_{\mathbb{A}}(t, a_t) \wedge mem_{\mathbb{A}}(u, a_u) \wedge \\
v = conc_{or}(t, u) \wedge a_v = abs_{or}(a_t, a_u) \wedge \neg(mem_{\mathbb{A}}(v, a_v))
\end{aligned} \tag{7}
$$

Validation of Manifested Soundness Violations. The programs generated by our approach for bugs with known CVEs were similar to the proof-of-concept implementations found in these CVEs. For previously unknown bugs, we logged the kernel verifier's state as it analyzes eBPF programs and also executed the eBPF program with the concrete operands produced by the SMT solver. We compared the parameters in the SMT solver's model and those from the kernel verifier and run-time result. This process entailed manually compiling and booting into each kernel version that we check, and running the generated programs. For the manifested bugs, we found exact agreement between the SMT model and the observed behaviors in all cases we checked.

5 C to Logic for Kernel's Abstract Operators

To prove the soundness of the kernel's abstract operators, we first have to extract the input-output semantics of the operators from the kernel's implementation in C into first-order logic. It is tedious and error-prone to manually write down the formulas for each version of the kernel. Further, the verifier's abstract semantics can change across versions. Hence, we automatically generate the first-order logic formula (in SMT-LIB format) directly from the verifier's C source code. Modeling C code in general is hard [42,46,64]. However, we observe that it is sufficient to handle a subset of C for the verifier's value-tracking routines.

Verifier's C Code for Value-tracking. The kernel uses two integers to represent abstract values for each of the five domains (Sect. 3). These 10 integers are encapsulated in a structure named `bpf_reg_state` (`reg_st` for short). The `tnum`

domain is further encapsulated within `reg_st` in a struct called `tnum`. This static "register state" is maintained for each register in the eBPF program being analyzed. The kernel has a single top-level function called `adjust_scalar_min_max_vals` (`adjust_scalar` for short) that is called for each abstract operator corresponding to ALU instructions [16]. This function takes three arguments: opcode and two register states named `dst` and `src` that track the abstract value in the destination and source register of the eBPF instruction, respectively. Depending on the opcode, one of several switch-cases is executed, which leads to instruction-specific function calls that modify the abstract values in `dst` and `src`. None of the functions updating register state in the call-chain have recursion or loops. The kernel has a structured way of accessing the members of `reg_st`. We use these specific features to translate C code to logic. The structures of the corresponding functions for jumps (`reg_set_min_max` and descendants) are similar.

Preprocessing the Verifier's C Code. We use the LLVM compiler's [47] intermediate representation (IR) because it allows us to handle complex C code and provides a collection of tools to modify, optimize, and analyze the IR. Figure 4(a) shows an overview of our tool's pipeline. Consider the case where we want to generate the SMT-LIB file for the abstract operator corresponding to the 32-bit bitwise OR instruction (`bpf_or32`). After obtaining the verifier's code in IR (stage ①), we proceed to apply our custom IR-transforming passes (stage ②). First, we remove functions that are not relevant to our purpose because they do not modify register state. Next, we inline all the function calls that `adjust_scalar` makes. Inlining is possible because there are no recursive functions or loops in the call-graph. Next, we need to create a slice of the verifier that is only concerned with `bpf_or32`. We inject an LLVM instruction in the `entry` basic block of `adjust_scalar` which sets the opcode to `bpf_or32`. LLVM's optimizer removes all irrelevant code from this IR with constant propagation and dead-code elimination. Next, we adapt a transformation pass from Seahorn's [42] codebase, which allows us to lower `memcpy` instructions to a sequence of stores. The result is a single function in LLVM IR, which captures the action of the abstract operator given input abstract states (i.e., `dst` and `src`) for one instruction (`bpf_or32`).

The LLVMToSMT Pass. In step ③, we use the theory of bitvectors to generate the first-order logic formula for the function obtained from step ②. Since we encode everything with bitvectors, we need a memory model to capture memory accesses. We model memory as a set of two disjoint regions pointed to by `dst` and `src`. Given that the memory is only accessed via the structure `reg_st`'s fields, we can further view memory as a set of named registers. This allows us to model the entire memory as a tree of bitvectors: the leaf nodes store bitvectors corresponding to the first-class members of `reg_st` (e.g. for `u64_min`), the non-leaf nodes store trees of aggregate types (e.g. for `tnum`). C `struct` member accesses in IR begin with a `getelementptr` (`GEP`) instruction, which calculates the pointer (address) of the struct's member. We use an indexing similar to that used by `GEP` to to identify the bitvector that corresponds to the accessed member.

Handling Straight Line Code and Branches. LLVM's IR is already in SSA form. Every IR instruction that produces a value defines a new temporary virtual

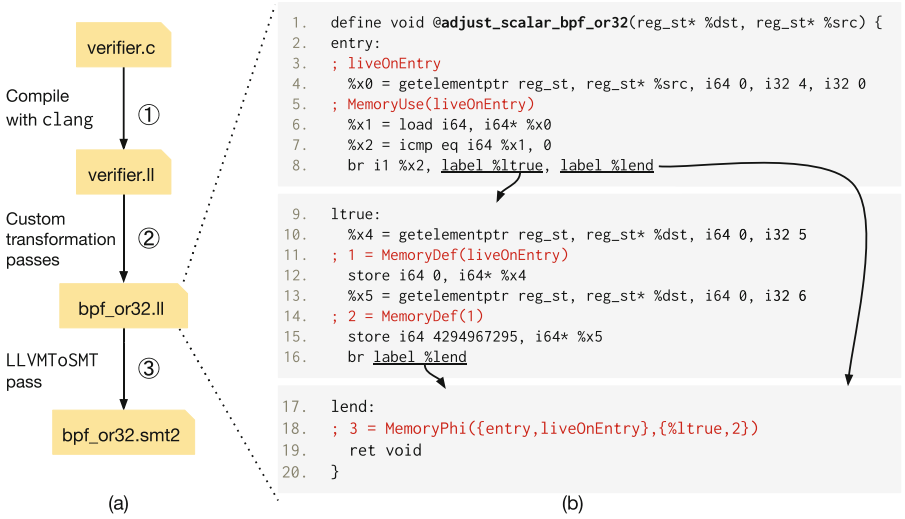

Fig. 4. (a) The pipeline for automatically generating an SMT-LIB file from the Linux kernel's `verifier.c`. Shown here is an instance of the pipeline for the `bpf_or32` instruction. (b) The LLVM IR presented as a CFG, overlaid with MemorySSA analysis in red, for a function `adjust_scalar_bpf_or32` that is representative of verifier code for `bpf_or32`. It takes as input two structs `dst` and `src` and modifies them.

register. We create a fresh bitvector variable when we encounter a temporary in the IR. Consider a simple addition instruction: `%y = add i64 %x, 3`. To encode the instruction, we create a formula that asserts an equality between a fresh bitvector BV_y and the existing one BV_x, based on the semantics of the instruction: $BV_y == BV_x + BV_{const3}$.

To handle branches, we precondition the SMT formula for each basic block with its path condition. As the IR we analyze does not contain loops, the control flow graph (CFG) is a directed acyclic graph. Hence, the path condition of each basic block is a disjunction of path conditions flowing through each incoming edge into the node corresponding to that block in the CFG. Phi nodes (ϕ's) in SSA merge the values flowing in from various paths. We use the `phi` instructions in IR to merge incoming values. We calculate an "edge condition" formula for each incoming edge to the `phi`. Then, we encode the `phi` instruction by appropriately setting the bitvector to the incoming values based on the edge condition.

Handling Memory Access Instructions. Our tool leverages LLVM's MemorySSA analysis [17] to handle `loads` and `stores`. The MemorySSA pass creates new versions of memory upon stores and branch merges, associates load instructions with specific versions, and provides a memory dependence graph between the memory versions. Figure 4 (b) shows an example CFG in IR overlaid with MemorySSA analysis (red). We maintain a one-to-one mapping between the different versions of memory presented by MemorySSA, and versions of our memory model consisting of bitvector-trees. `liveOnEntry` (line 3) is the memory version

at the start of the function. The bitvectors in the corresponding bitvector-tree are the input operands for the kernel's abstract operators.

Every load instruction is annotated with a `MemoryUse` (e.g.. the load instruction on line 6 reads from the `liveOnEntry` memory version), and preceded by a `GEP`. Thus, we choose the appropriate bitvector-tree and index into it to obtain the appropriate bitvector (say BV_{src0}). We encode the load instruction as: $(BV_{x1} == BV_{src0})$. A store instruction (e.g. line 12, annotated using a `MemoryDef`) modifies an existing memory version (`liveOnEntry`) to create new version (`1`). We create a new bitvector-tree and map it to version `1`. The bitvectors in this bitvector-tree are exactly the same as `liveOnEntry`'s, except for the bitvector in the location that the store modifies. The latter bitvector is replaced with the bitvector mapped to the temporary used for the store. For a `MemoryPhi` node (e.g. line 18, creating version `3`), we create a new bitvector-tree for the latest memory version (e.g. `3`). Similar to regular `phi` nodes, we use the edge condition of the incoming edges to conditionally set each bitvector in the new bitvector-tree to the corresponding bitvector in the memory version propagated through that edge.

The bitvector-tree corresponding to the active memory version at the point of the (unique) `ret` instruction (e.g. `3` in the `lend` block) contains the output operands for the kernel's abstract operators.

6 Experimental Evaluation

Our prototype, Agni [18,72], automatically checks the soundness of the value tracking algorithms in various versions of the kernel eBPF verifier. It uses LLVM 12 [47] for the C to logic translation and the Z3 SMT solver [36] for checking formulas. The source code for our prototype is publicly available [18,72]. We evaluate Agni to determine the effectiveness in checking soundness of the kernel verifier and the ability to generate eBPF programs that manifest soundness violations (which we call proof-of-concepts, or POCs).

Checking Soundness Across Kernel Versions. We have automatically checked the soundness of all combinations of abstract operators and abstract domains for kernels between versions 4.14 and 5.19. Figure 5(a) provides a summary of our results. To keep the size of the table short, we only report kernel versions starting from 4.14 that are known to have a documented CVE or a bug that is distinct from one in a prior kernel version (4.14, 5.5, 5.7-rc1, 5.8, ...). We evaluated intermediate kernel versions that are not reported; our tool can support all kernel versions between 4.14 to 5.19 (the latest as of this writing).

We compare our generic soundness specification (Sect. 4.1, labeled *gen* in columns 2,4,6) and the refined one (Sect. 4.2, labeled *sro* in columns 3,5,7). A kernel with at least one potentially unsound domain or operator is considered unsound (columns 2 and 3). Operator+domain pairs that violated the soundness specification are reported in columns 4 and 5. Those operators that violated soundness in at least one domain are reported in columns 6 and 7.

All kernel versions including the latest ones are unsound with respect to the generic soundness specification (column 2). Even in one of the latest ver-

Kernel Version	Sound?		Num. of Violations		Num. of Unsound Operators	
	gen	*sro*	*gen*	*sro*	*gen*	*sro*
4.14	✗	✗	23	21	9	7
5.5	✗	✗	32	30	12	10
5.7-rc1	✗	✗	101	99	31	31
5.7	✗	✗	69	67	15	15
5.8	✗	✗	69	67	15	15
5.9	✗	✗	67	65	15	15
5.10	✗	✗	74	65	17	17
5.10-rc1	✗	✗	74	71	17	17
5.11	✗	✗	71	62	16	16
5.12	✗	✗	71	62	16	16
5.13	✗	✓	9	0	6	0
5.14	✗	✓	9	0	6	0
5.16	✗	✓	9	0	6	0
5.17	✗	✓	9	0	6	0
5.18	✗	✓	9	0	6	0
5.19	✗	✓	9	0	6	0

(a)

Kernel Version	Num. of Total Violations	All POCs Synth?	Program Length		
			1	2	3
4.14	21	✗	14	4	0
5.5	30	✗	0	20	2
5.7-rc1	99	✓	55	44	0
5.7	67	✓	39	28	0
5.8	67	✓	39	28	0
5.9	65	✓	39	26	0
5.10	65	✓	19	44	2
5.10-rc1	71	✓	39	32	0
5.11	62	✓	16	44	2
5.12	62	✓	16	44	2

(b)

Fig. 5. (a) Soundness violations detected with the generic soundness specification (Sect. 4.1, labeled *gen*) in comparison to the refined specification (Sect. 4.2, labeled *sro*). We show the number of violating operator+domain pairs (columns 4-5) and number of unsound operators (columns 6-7) (b) Number of generated POCs and their lengths for unsound operator+domains after *sro* checks.

sions of the kernel (v5.19), 6 operators corresponding to `bpf_xor64`, `bpf_xor32`, `bpf_and64`, `bpf_or64`, `bpf_or32`, and `bpf_and32` are unsound according to the generic soundness specification (column 6, row of kernel version 5.19). Refining the soundness specification enables us to prove the soundness of all operators in kernels newer than 5.13 (column 3). However, even the latter reports violations for older kernels. Among those violations, 27 were previously unknown. A single wrong abstract operator can violate the soundness of many abstract domains (up to 5). The refined (*sro*) specification reduces the reported soundness violations by $\approx 6.8\%$ in potentially unsound kernel versions and by 100% in sound ones.

We observed that the 64-bit jump instructions and 64-bit/32-bit bitwise instructions exhibited the largest number of soundness violations. The unsoundness persisted across multiple kernel versions (until eventually patched).

Generating POCs for Unsound kernels. We evaluate the ability of differential synthesis (Sect. 4.3) to generate eBPF programs that manifest soundness bugs. Figure 5(b) summarizes our results. Starting with operator+domain pairs from soundness violations uncovered by *sro* (column 2), we report whether all operator+domain violations were successfully manifested using POCs (column 3) and the lengths of the POCs successfully generated (columns 4,5,6). We produced a POC for $\approx 97\%$ of soundness violations across kernel versions (validated as described in Sect. 4.3). The smallest POCs for many violations require multi-instruction programs. For example, none of the soundness violations in version 5.5 may be manifested with a single eBPF instruction. We generated a POC for all soundness violations for all but 2 versions of the kernel (for versions 4.14 and 5.5, we generated a POC for all but 3 and 8 violations respectively). The ability to manifest almost all of the reported *sro* violations speaks to the significance and precision of the refinement in the soundness specification. Our differential

synthesis technique may enable developers to experiment with concrete eBPF programs to validate and debug unsound behaviors in the kernel verifier.

Some bugs in the eBPF verifier are well known security vulnerabilities and have known POCs [51,62]. We have generated a POC, of equal or lesser size, for all known CVEs in the kernel versions analyzed. For example, we have generated a POC for a well known bug with two instructions instead of four [62].

Time Taken to Verify kernels and Generate POCs. We conducted our experiments on the Cloudlab [37] testbed, using a machine with two 10-core Intel Skylake CPUs running at 2.20 GHz with 192 GB of memory. When using the generic soundness specifications, 90% of the abstract operators (eBPF instructions) were checked for soundness within $\approx$ 100 minutes. If deemed unsound, the refined specification was checked in $\approx$ 30 minutes for $\approx$ 90% of the unsound operators. On the extreme, verifying some operators, as well as finding a POC for some soundness violations, may take a long time (2000 min or more). We attribute this to the significant size of the SMT-LIB formulas that are generated. We were able to find POCs for 90% of the soundness violations in kernel versions 5.7-rc1 through 5.12 within a few hours.

7 Limitations and Caveats

The results in this paper must be interpreted with the following caveats.

Only Range Analysis is Considered. There are other static analyses in the kernel verifier beyond range analysis (Sect. 1). These include tracking register liveness for reading and writing, and detecting speculative execution vulnerabilities.

Coverage of eBPF Abstract Operators. We exclude verifying the soundness of the abstract operators corresponding to multiplication as they cause our SMT verifications to time out. This is primarily due to the presence of 64-bit bitvector multiplication in the SMT encoding of these operators. We have verified their soundness using 8-bit bitvectors. Our results on (un)soundness cover all other abstract arithmetic, logic, and branching operators (Sect. 4.1).

Trusted Computing Base. Our C to SMT translation (Sect. 5) and soundness proofs have software dependencies including the LLVM compiler infrastructure, the Z3 solver, and our translation passes, which together form our trusted computing base. We have unit tested our C-to-SMT translations extensively. We validated our synthesized POCs by manually executing them in Linux kernels running inside the QEMU emulator, replicating the soundness bugs. Despite our best efforts, it is possible that there are bugs in our software infrastructure.

Incompleteness of Differential Synthesis. The differential synthesis approach is incomplete (Sect. 4.3). If our refined verification condition (Eq. (4)) finds an operator unsound, and the synthesis is unable to produce a POC, there are two possibilities. First, there may be long programs which could manifest the unsound behavior. Our enumerative algorithm currently times out for programs of length ≥ 4. Second, it is possible that the bug cannot be manifested with any concrete eBPF program, and is reported due to overapproximation in the soundness specification.

8 Related Work

Closest Related Work. The two closely related prior works are: (1) a paper on `tnum` verification [71], and (2) a recent manuscript on verifying range analysis [21]. The `tnum` paper explores formal verification for a single abstract domain: `tnums`. The recent manuscript [21] also aims to prove the soundness of the eBPF verifier's value-tracking. In contrast, our work differs by (1) exposing the non-modular nature of the abstract operators in the kernel, and (2) proposing a method to reason about abstract operators for both arithmetic and branches, (3) automatically generating VCs from kernel source code, and (4) synthesizing eBPF programs that exercise the divergence of abstract and concrete semantics.

Safety of eBPF Programs And Static Analyzers. eBPF compilation and interpreter safety has been a site of recent endeavors [59,60,69,73,74]. PREVAIL [39] uses abstract interpretation using the zone abstract domain for checking safety outside the kernel. In contrast, we focus on proving the soundness of the in-kernel verifier.

Abstract Interpretation And Domain Refinement. Prior work on abstract interpretation [30,31,33] and value-tracking abstract domains [55,56,68] have indirectly influenced the eBPF verifier's design [61,71]. The idea of combining abstract domains to enhance the precision of abstract representations was first introduced by Cousot with the reduced product and disjunctive completion domain refinements [29,34] and further improved by others [70]. A systematic survey on product abstract operators is also available [28]. Specifically, we tailor our work to verify the abstract operators in the Linux kernel.

C to First-order Logic. Similar to our approach that generates first-order-logic formulas from C code, prior tools also generate verification conditions from C code [42,46,54,64]. A few of them, SMACK [64] and SeaHorn [42], use LLVM IR for this purpose. These tools support a rich subset of C. They typically model memory as a linear array of bytes, which is not ideal for modeling kernel source code. We explore a subset of C that is sufficient to handle kernel code and still generates queries using only the bitvector theory, which enables us to efficiently verify soundness for multiple versions of the kernel.

9 Conclusion

We present a fully automated method to verify the soundness of range analysis in the Linux kernel's eBPF verifier. We are able to check the soundness of multiple kernel versions automatically because we generate the verification conditions for the abstract operators directly from the kernel C code. We develop specifications for reasoning about soundness when multiple abstract domains are combined in a non-modular fashion in the kernel. Our refinement to this specification, capturing preconditioning in the kernel, proves the soundness of recent Linux kernels. We also successfully generate concrete eBPF programs

that demonstrate the divergence between abstract and concrete semantics when soundness checks fail. Our next step is to push for incorporating this approach in the kernel development process, to help eliminate verifier bugs during code review.

Acknowledgement. This paper is based upon work supported in part by the National Science Foundation under FMITF-Track I Grant No. 2019302. We thank the CAV reviewers, and He Zhu for their valuable feedback. We also thank CloudLab for providing the research testbed for our experiments.

References

1. bpf: fix incorrect sign extension in check_alu_op(). Accessed 14 Jan 2020. https://github.com/torvalds/linux/commit/95a762e2c8c942780948091f8f2a4f32fce1ac6f
2. bpf, x32: Fix bug with ALU64 LSH, RSH, ARSH BPF_X shift by 0. Accessed 14 Apr 2021. https://github.com/torvalds/linux/commit/68a8357ec15bdce55266e9fba8b8b3b8143fa7d2
3. CVE-2017-16996 Mishandling of register truncation. Accessed 22 Jan 2023. https://nvd.nist.gov/vuln/detail/CVE-2017-16996
4. CVE-2017-17852 Mishandling of 32-bit ALU ops. Accessed 22 Jan 2023. https://nvd.nist.gov/vuln/detail/CVE-2017-17852
5. CVE-2017-17853 Mishandling of 32-bit ALU ops. Accessed 22 Jan 2023. https://nvd.nist.gov/vuln/detail/CVE-2017-17853
6. CVE-2017-17864 Mishandled comparison between pointer and unknown data types. Accessed 14 Jan 2020. https://nvd.nist.gov/vuln/detail/CVE-2017-17864
7. CVE-2018-18445 Mishandling of 32-bit RSH op. Accessed 22 Jan 2023. https://nvd.nist.gov/vuln/detail/CVE-2018-18445
8. CVE-2020-8835 Mishandling of bounds tracking for 32-bit JMPs. Accessed 22 Jan 2023. https://nvd.nist.gov/vuln/detail/CVE-2020-8835
9. 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. Accessed 22 Jan 2023. CVE-2021-3490
10. Facebook's Katran load balancer: Kernel XDP program. Accessed 14 Jan 2020. https://github.com/facebookincubator/katran/blob/master/katran/lib/bpf/balancer_kern.c
11. Linux BPF verifier. Accessed 14 Jan 2020. https://github.com/torvalds/linux/blob/master/kernel/bpf/verifier.c
12. Netconf 2018 day 1. Accessed 19 Jan 2020. https://lwn.net/Articles/757201/
13. BPF instruction set. Accessed 14 Jan 2020. https://github.com/iovisor/bpf-docs/blob/master/eBPF.md (2017)
14. Linux verifier's abstract u64 addition (kernel v6.0). Accessed 08 Nov 2022. https://github.com/torvalds/linux/blob/v6.0/kernel/bpf/verifier.c#L8333 (2022)
15. Linux verifier's abstract u64 bitwise AND (kernel v6.0). Accessed 08 Nov 2022. https://github.com/torvalds/linux/blob/v6.0/kernel/bpf/verifier.c#L8513 (2022)
16. Linux verifier's top-level function for value-tracking in scalar for alu instructions (kernel v6.0): adjust_scalar_min_max_vals: Accessed 27 Jan 2023. https://github.com/torvalds/linux/blob/90aaef4e35c4a74b0f1593d06e39eda867ef13d3/kernel/bpf/verifier.c#L10524 (2023)

17. LLVM's MemorySSA. Accessed 27 Jan 2023. https://llvm.org/docs/MemorySSA.html (2023)
18. Verifying the Verifier: eBPF Range Analysis Verification (2023). https://doi.org/10.5281/zenodo.7931901
19. Alur, R., Singh, R., Fisman, D., Solar-Lezama, A.: Search-based program synthesis. Commun. ACM **61**(12), 84–93 (2018). https://doi.org/10.1145/3208071
20. Bansal, S., Aiken, A.: Automatic generation of peephole superoptimizers. SIGOPS Oper. Syst. Rev. **40**(5), 394–403 (2006). https://doi.org/10.1145/1168917.1168906
21. Bhat, S., Shacham, H.: Formal verification of the linux kernel eBPF verifier range analysis. Accessed 27 Jan 2023. https://sanjit-bhat.github.io/assets/pdf/ebpf-verifier-range-analysis22.pdf (2022)
22. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49059-0_14
23. Borkmann, D.: bpf: Fix scalar32_min_max_or bounds tracking. Accessed 6 Nov 2022. https://github.com/torvalds/linux/commit/5b9fbeb75b6a98955f628e205ac26689bcb1383e (2020)
24. Borkmann, D.: bpf: Undo incorrect __reg_bound_offset32 handling. Accessed 6 Nov 2022. https://git.kernel.org/pub/scm/linux/kernel/git/netdev/net-next.git/commit/?id=f2d67fec0b43edce8c416101cdc52e71145b5fef (2020)
25. Borkmann, D.: bpf: Fix alu32 const subreg bound tracking on bitwise operations. Accessed 6 Nov 2022. https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf.git/commit/?id=049c4e13714ecbca567b4d5f6d563f05d431c80e (2021)
26. Borkmann, D.: bpf: Fix signed_sub, add32_overflows type handling. Accessed 6 Nov 2022. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=bc895e8b2a64e502fbba72748d59618272052a8b (2021)
27. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7
28. Cortesi, A., Costantini, G., Ferrara, P.: A survey on product operators in abstract interpretation. Electr. Proceed. Theoret. Comput. Sci. **129**, 325–336 (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 (ICCL1994), pp. 95–112 (1994). https://doi.org/10.1109/ICCL.1994.288389
30. Cousot, P.: Abstract interpretation based formal methods and future challenges. In: Wilhelm, R. (ed.) Informatics. LNCS, vol. 2000, pp. 138–156. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44577-3_10
31. Cousot, P.: Lecture 13 notes: MIT 16.399, abstract interpretation. Accessed 16 Apr 2021. 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, pp. 238–252. POPL 1977, 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 1979, 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. Accessed 6 Nov 2022. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=4374f256ce8182019353c0c639bb8d0695b4c941 (2017)
36. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
37. Duplyakin, D., et al.: The design and operation of cloudLab. In: Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference, pp. 1–14. USENIX ATC 2019, USENIX Association, USA (2019)
38. Fabre, A.: L4Drop: XDP DDoS mitigations. Accessed 19 Jan 2020. https://blog.cloudflare.com/l4drop-xdp-ebpf-based-ddos-mitigations/
39. Gershuni, E., et al.: Simple and precise static analysis of untrusted Linux Kernel extensions. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 1069–1084. PLDI 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3314221.3314590
40. Granger, P.: Improving the results of static analyses of programs by local decreasing iterations. In: Shyamasundar, R. (ed.) FSTTCS 1992. LNCS, vol. 652, pp. 68–79. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-56287-7_95
41. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. SIGPLAN Not. **46**(6), 62–73 (2011). https://doi.org/10.1145/1993316.1993506
42. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification, pp. 343–361. Springer International Publishing, Cham (2015)
43. Horn, J.: Arbitrary read+write via incorrect range tracking in ebpf. Accessed 19 Jan 2020. https://bugs.chromium.org/p/project-zero/issues/detail?id=1454
44. Horn, J.: BPF: fix 32-bit ALU op verification. Accessed 6 Nov 2022. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=468f6eafa6c44cb2c5d8aad35e12f06c240a812a (2017)
45. Horn, J.: bpf: 32-bit RSH verification must truncate input before the ALU op. Accessed 6 Nov 2022. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=b799207e1e1816b09e7a5920fbb2d5fcf6edd681 (2018)
46. Kroening, D., Tautschnig, M.: CBMC – C bounded model checker. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 389–391. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_26
47. Lattner, C., Adve, V.: Llvm: a compilation framework for lifelong program analysis & transformation. In: International symposium on code generation and optimization, 2004. CGO 2004, pp. 75–86. IEEE (2004). https://doi.org/10.1109/CGO.2004.1281665
48. Linux eBPF maintainers: bounds syncing for abstract registers. Accessed 31 Jan 2023. https://github.com/torvalds/linux/blob/v6.0/kernel/bpf/verifier.c#L1565 (2023)
49. Linux eBPF maintainers: using bounds syncing at end of alu operations. Accessed 31 Jan 2023. https://github.com/torvalds/linux/blob/v6.0/kernel/bpf/verifier.c#L9016 (2023)

50. Lucas Leong: ZDI-20-1440: An incorrect calculation bug in the Linux Kernel eBPF verifier. Accessed 22 Jan 2023. https://www.zerodayinitiative.com/blog/2021/1/18/zdi-20-1440-an-incorrect-calculation-bug-in-the-linux-kernel-ebpf-verifier
51. Manfred Paul: CVE-2020-8835: Linux kernel privilege escalation via improper eBPF program verification. Accessed 22 Jan 2023. https://www.zerodayinitiative.com/blog/2020/4/8/cve-2020-8835-linux-kernel-privilege-escalation-via-improper-ebpf-program-verification
52. Massalin, H.: Superoptimizer: A look at the smallest program. In: Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems, pp. 122–126. ASPLOS II, Association for Computing Machinery, New York, NY, USA (1987). https://doi.org/10.1145/36206.36194
53. McCanne, S., Jacobson, V.: The BSD packet filter: a new architecture for user-level packet capture. In: USENIX Winter 1993 Conference (USENIX Winter 1993 Conference). USENIX Association, San Diego, CA (1993). https://www.usenix.org/conference/usenix-winter-1993-conference/bsd-packet-filter-new-architecture-user-level-packet
54. Merz, F., Falke, S., Sinz, C.: LLBMC: bounded model checking of C and C++ programs using a compiler IR. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 146–161. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27705-4_12
55. Miné, A.: Abstract domains for bit-level machine integer and floating-point operations. In: WING2012 - 4th International Workshop on invariant Generation, p. 16. Manchester, United Kingdom (2012). https://hal.science/hal-00748094
56. Miné, A.: Tutorial on static inference of numeric invariants by abstract interpretation. Found. Trends® Programm. Lang. **4**(3–4), 120–372 (2017). https://doi.org/10.1561/2500000034
57. 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
58. Mukherjee, M., Kant, P., Liu, Z., Regehr, J.: Dataflow-based pruning for speeding up superoptimization. Proc. ACM Program. Lang. **4**, 3428245 (OOPSLA) (2020). https://doi.org/10.1145/3428245
59. 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, pp. 225–242. SOSP 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3341301.3359641
60. 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. OSDI2020, USENIX Association, USA (2020)
61. Onderka, J., Ratschan, S.: Fast three-valued abstract bit-vector arithmetic. In: Finkbeiner, B., Wies, T. (eds.) VMCAI 2022. LNCS, vol. 13182, pp. 242–262. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-94583-1_12
62. Palmiotti, V.: Kernel pwning with eBPF: a love story. Accessed 31 August 2021. https://www.graplsecurity.com/post/kernel-pwning-with-ebpf-a-love-story
63. Phothilimthana, P.M., Thakur, A., Bodik, R., Dhurjati, D.: Scaling up superoptimization. In: Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 297–310. ASPLOS 2016, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2872362.2872387

64. Rakamarić, Z., Emmi, M.: SMACK: decoupling source language details from verifier implementations. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 106–113. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_7
65. 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, pp. 34–43. LCTES 2006, Association for Computing Machinery, New York, NY, USA (2006). https://doi.org/10.1145/1134650.1134657
66. Rick Larabee: eBPF and Analysis of the get-rekt-linux-hardened.c Exploit for CVE-2017-16995. Accessed 22 Jan 2023. https://ricklarabee.blogspot.com/2018/07/ebpf-and-analysis-of-get-rekt-linux.html
67. Sasnauskas, R., Chen, Y., Collingbourne, P., Ketema, J., Taneja, J., Regehr, J.: Souper: a synthesizing superoptimizer. CoRR abs/1711.04422 (2017). https://doi.org/10.48550/arXiv.1711.04422
68. Singh, G., Püschel, M., Vechev, M.: Fast polyhedra abstract domain. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, pp. 46–59. POPL 2017, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3009837.3009885
69. 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.) CAV 2020. LNCS, vol. 12225, pp. 564–586. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_29
70. Venet, A.: Abstract cofibered domains: Application to the alias analysis of untyped programs. In: Cousot, R., Schmidt, D.A. (eds.) SAS 1996. LNCS, vol. 1145, pp. 366–382. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61739-6_53
71. Vishwanathan, H., Shachnai, M., Narayana, S., Nagarakatte, S.: Sound, precise, and fast abstract interpretation with tristate numbers. In: Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization, pp. 254–265. CGO 2022, IEEE Press (2022). https://doi.org/10.1109/CGO53902.2022.9741267
72. Vishwanathan, H., Shachnai, M., Narayana, S., Nagarakatte, S.: Agni: verifying the Verifier (eBPF Range Analysis Verification). Accessed 29 May 2023. https://github.com/bpfverif/ebpf-range-analysis-verification-cav23 (2023)
73. 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, pp. 33–47. OSDI2014, USENIX Association, USA (2014)
74. 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, pp. 50–64. SIGCOMM 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3452296.3472929