Sound, Precise, and Fast Abstract Interpretation with Tristate Numbers

Harishankar Vishwanathan*, Matan Shachnai†, Srinivas Narayana‡, and Santosh Nagarakatte§
Rutgers University, USA
*harishankar.vishwanathan@rutgers.edu, †mys35@cs.rutgers.edu, ‡srinivas.narayana@rutgers.edu,
§santosh.nagarakatte@cs.rutgers.edu

***Abstract*—Extended Berkeley Packet Filter (BPF) is a language and run-time system that allows non-superusers to extend the Linux and Windows operating systems by downloading user code into the kernel. To ensure that user code is safe to run in kernel context, BPF relies on a static analyzer that proves properties about the code, such as bounded memory access and the absence of operations that crash. The BPF static analyzer checks safety using abstract interpretation with several abstract domains. Among these, the domain of tnums (tristate numbers) is a key domain used to reason about the bitwise uncertainty in program values. This paper formally specifies the tnum abstract domain and its arithmetic operators. We provide the first proofs of soundness and optimality of the abstract arithmetic operators for tnum addition and subtraction used in the BPF analyzer. Further, we describe a novel sound algorithm for multiplication of tnums that is more precise and efficient (runs 33% faster on average) than the Linux kernel's algorithm. Our tnum multiplication is now merged in the Linux kernel.**

***Index Terms*—Abstract domains, Program verification, Static analysis, Kernel extensions, eBPF**

I. Introduction

Static analysis is an integral part of compilers [1, 2, 3, 4], sandboxing technologies [5, 6, 7], and continuous integration testing [8]. For example, static analysis may be used to prove that the value of a program variable will always be bounded by a known constant, allowing a compiler to eliminate dead code [9] or a sandbox to remove an expensive run-time check [5].

Our work is motivated by static analysis in the context of *Berkeley Packet Filter (BPF)*, a language and run-time system [10, 11] that enables users to extend the functionality of the Linux and Windows operating systems without writing kernel code. BPF is widely deployed in production systems today [12, 13, 14, 15, 16, 17, 18]. BPF uses a static analyzer to validate that user programs are *safe* before they are executed in kernel context [11, 7]: the analyzer must be able to show that the program does not access unpermitted memory regions, does not leak privileged kernel data, and does not crash. If the analyzer is unable to prove these properties, the user program is *rejected* and cannot execute in kernel context.

BPF static analysis must be *sound, precise,* and *fast*.

- *Soundness:* Unsound analysis that accepts malicious code may result in arbitrary read-write capabilities for users in the kernel [19]. Unfortunately, the Linux static analyzer has been a source of numerous such bugs in the past [20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33].
- *Precision:* To provide a usable system, the analyzer must not reject safe programs due to imprecision in its analysis. Users often need to rewrite their programs to get their code past the analyzer [7, 34, 35].
- *Speed:* The analyzer must keep the time and overheads to load a BPF program minimal [11, 36, 37]. Programs are often used to trace systems running heavy workloads.

The BPF static analyzer employs abstract interpretation [38] with multiple abstract domains to track the types, liveness, and values of program variables across all executions. One of the key abstract domains, termed *tristate numbers* or *tnums* in the Linux kernel [39], tracks which bits of a value are known to be 0, known to be 1, or unknown (denoted μ) across executions. For example, a 4-bit variable x abstracted to $01\mu0$ can take on the binary values 0100 and 0110. The analyzer can infer that the expression $x \leq 8$ will always return $true$, and use this fact later to show the safety of a memory access.

The kernel provides algorithms to implement bit-wise operations such as and (&), or (|), and shifts (<<, >>) over tnums. The kernel also provides efficient algorithms for arithmetic (addition, subtraction, and multiplication) over tnums. In particular, addition and subtraction run in $O(1)$ time over n-bit program variables given n-bit machine arithmetic instructions.

Unfortunately, the kernel provides no formal reasoning or proofs of soundness or precision of its algorithms. Prior works that explored abstract domains for bit-level reasoning [40, 41, 42, 43, 3, 44] provide sound and precise abstract operators for bit-wise operations (&, |, >>, *etc.*). The only arithmetic algorithms we are aware of [42] are much slower than the kernel's algorithms (§II). Arithmetic operations are tricky to reason about as they propagate uncertainty across bits in non-obvious ways. For example, suppose a is known to be the n-bit constant $11\cdots1$ and b is either 0 or 1 across all executions. Only one bit is uncertain among the operands, yet all bits in $a+b$ are unknown, since $a+b$ can be either $11\cdots1$ or $00\cdots0$.

This paper makes the following contributions (§III). We provide the first proofs of *soundness* and *optimality* (*i.e.,* maximal precision [3, 45]) of the kernel's algorithms for addition and subtraction. We believe this result is remarkable for abstract operators exhibiting $O(1)$ run time and reasoning about uncertainty across bits. We were unable to prove the soundness of the kernel's tnum multiplication. Instead, we present a novel multiplication algorithm that is provably sound. It is also more precise and 32% faster than prior implementations [42, 39].

978-1-6654-0584-3/22 © 2022 IEEE

Accepted for publication by IEEE. © 2022 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

This algorithm is now merged into the latest Linux kernels. Our reproducible artifact is publicly available [46].

II. Background

The BPF static analyzer in the kernel checks the safety of BPF programs by performing abstract interpretation using the tnum abstract domain (among others). In this section, we provide a primer on abstract interpretation and describe the tnum abstract domain and its operators.

A. Primer on Abstract Interpretation

Abstract interpretation [38] is a form of static analysis that captures the values of program variables in all executions of the program. Abstract interpretation employs *abstract values* and *abstract operators*. Abstract values are drawn from an *abstract domain*, each element of which is a concise representation of a set of concrete values that a variable may take across executions. For example, an abstract value from the interval abstract domain [47] $\{[a, b] \mid a, b \in \mathbb{Z}, a \leq b\}$ models the set of all concrete integer values (*i.e.,* $x \in \mathbb{Z}$) such that $a \leq x \leq b$.

Abstraction and Concretization functions. An *abstraction function* α takes a concrete set and produces an abstract value, while a *concretization function* γ produces a concrete set from an abstract value. For example, the abstraction of the set $\{2, 4, 5\}$ in the interval domain is $[2, 5]$, which produces the set $\{2, 3, 4, 5\}$ when concretized.

Formally, the domains of the abstraction and concretization functions are two partially-ordered sets (posets) that induce a lattice structure. We denote the concrete poset $\mathbb{C}$ with the ordering relationship among elements $\sqsubseteq_{\mathbb{C}}$. Similarly, we denote the abstract poset $\mathbb{A}$ with the ordering relationship $\sqsubseteq_{\mathbb{A}}$. For example, the interval domain employs the concrete poset $\mathbb{C} \triangleq 2^{\mathbb{Z}}$, the power set of $\mathbb{Z}$, with the subset relation $\subseteq$ (*e.g.,* $\{1, 2\} \subseteq \{1, 2, 3\}$) as its ordering relation. The abstract poset is $\mathbb{A} \triangleq \mathbb{Z} \times \mathbb{Z}$ with the ordering relation $[a, b] \sqsubseteq_{\mathbb{A}} [c, d] \Leftrightarrow (c \leq a) \wedge (d \geq b)$.

A value $a \in \mathbb{A}$ is a *sound* abstraction of a value $c \in \mathbb{C}$ if and only if $c \sqsubseteq_{\mathbb{C}} \gamma(a)$. Moreover, a is an *exact* abstraction of c if $c = \gamma(a)$. Abstractions are often not exact, over-approximating the concrete set to permit concise representation and efficient analysis in the abstract domain. For example, the interval $[2, 5]$ is a sound but inexact abstraction of the set $\{2, 4, 5\}$.

Abstract operators are functions over abstract values which return abstract values. An abstract operator implements an "abstract version" of a concrete operation over concrete sets, hence enabling a static analysis to construct the abstract results of program execution. For example, abstract integer addition in the interval domain (denoted $+_{\mathbb{A}}$) abstracts concrete integer addition (denoted $+_{\mathbb{C}}$) as follows: $[a_1, b_1] +_{\mathbb{A}} [a_2, b_2] \triangleq [a_1 +_{\mathbb{C}} a_2, b_1 +_{\mathbb{C}} b_2]$. Abstract operators typically over-approximate the resulting concrete set to enable decidable and fast analysis at the expense of precision. For a concrete set $S \in \mathbb{C}$, suppose we use the shorthand $f(S)$ to denote the set $\{f(x) \mid x \in S\}$. An abstract operator $g : \mathbb{A} \rightarrow \mathbb{A}$ is a *sound* abstraction of a concrete operator $f : \mathbb{C} \rightarrow \mathbb{C}$ if $\forall a \in \mathbb{A} : f(\gamma(a)) \sqsubseteq_{\mathbb{C}} \gamma(g(a))$. Further, g is *exact* if $\forall a \in A : f(\gamma(a)) = \gamma(g(a))$.

Galois connection. Pairs of abstraction and concretization functions (α, γ) are said to form a Galois connection if [45]:

1) α is monotonic, *i.e.,* $x \sqsubseteq_{\mathbb{C}} y \implies \alpha(x) \sqsubseteq_{\mathbb{A}} \alpha(y)$
2) γ is monotonic, $a \sqsubseteq_{\mathbb{C}} b \implies \gamma(a) \sqsubseteq_{\mathbb{A}} \gamma(b)$
3) $\gamma \circ \alpha$ is extensive, *i.e.,* $\forall c \in \mathbb{C} : c \sqsubseteq_{\mathbb{C}} \gamma(\alpha(c))$
4) $\alpha \circ \gamma$ is reductive, *i.e.,* $\forall a \in \mathbb{A} : \alpha(\gamma(a)) \sqsubseteq_{\mathbb{A}} a$

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.

Optimality. Suppose $(\mathbb{C}, \sqsubseteq_{\mathbb{C}}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathbb{A}, \sqsubseteq_{\mathbb{A}})$ is a Galois connection. Given a concrete operator $f : \mathbb{C} \rightarrow \mathbb{C}$, the abstract operator $\alpha \circ f \circ \gamma$ is the smallest sound abstraction of f: that is, for any sound abstraction $g : \mathbb{A} \rightarrow \mathbb{A}$ of f, we have $\forall a \in \mathbb{A} : \alpha(f(\gamma(a))) \sqsubseteq_{\mathbb{A}} g(a)$. We call $\alpha \circ f \circ \gamma$ the *optimal*, or maximally precise abstraction, of f.

B. The Tnum Abstract Domain

Tnums enable performing bit-level analysis by abstracting each bit of a program variable separately. Across executions, each bit is either known to be 0, known to be 1, or uncertain, denoted by μ. For an n-bit program variable, the abstract value corresponding to the variable has n ternary digits, or *trits*. Each trit has a value of 0, 1, or μ.

Bit-level abstract interpretation has been addressed in several prior works using the bitfield abstract domain [40, 41, 42] and the known bits abstract domain [43, 3, 44]. Abstraction and concretization functions forming a Galois connection already exist [41], as well as sound and optimal abstract operators for bit-level operations like bit-wise-and (`&`), bit-wise-or (`|`), and shifts (`<<`, `>>`) [3, 41]. In contrast to prior work, this paper explores provably sound, optimal, and computationally-efficient abstract operators corresponding to *arithmetic operations* such as addition, subtraction, and multiplication. The Linux kernel analyzer, despite heavily leveraging this domain's abstract operations, formally lays out neither the soundness nor optimality for the abstract arithmetic operations.

Abstract and Concrete Domains. Tnums track each bit of variables drawn from the set of n-bit integers $\mathbb{Z}_n$.

- The concrete poset is $\mathbb{C} \triangleq 2^{\mathbb{Z}_n}$, the power set of $\mathbb{Z}_n$. The ordering relation $\sqsubseteq_{\mathbb{C}}$ is the subset relation:

$$a \sqsubseteq_{\mathbb{C}} b \triangleq a \subseteq b \tag{1}$$

- The abstract poset $\mathbb{A}$ is the set of n-trit tnums $\mathbb{T}_n$ (each trit is 0, 1, or μ). Suppose we represent the trit in the ith position of a by $a[i]$. The ordering relation $\sqsubseteq_{\mathbb{A}}$ between abstract elements is defined by:

$$P \sqsubseteq_{\mathbb{A}} Q \triangleq \forall i, 0 \leq i \leq n-1, \forall k \in \{0, 1\} : \\ (P[i] = \mu \Rightarrow Q[i] = \mu) \wedge (Q[i] = k \Rightarrow P[i] = k) \tag{2}$$

Fig. 1 shows Hasse diagrams of the lattices induced by these posets for integers with bit width $n = 2$. The concrete domain consists of all elements of the power set of $\{0, 1, 2, 3\}$ and the abstract domain consists of tnums of the form $t_1 t_0$ where each

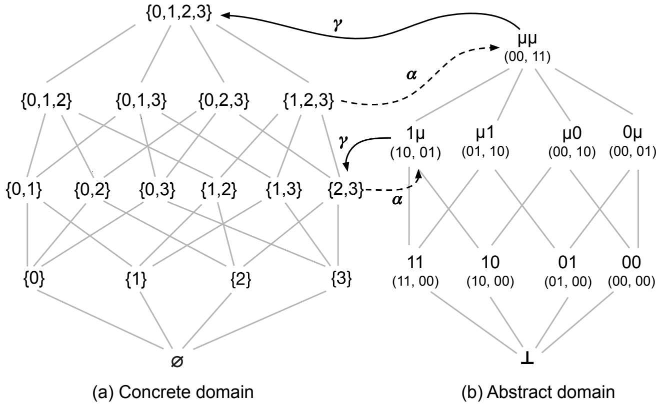

Fig. 1: Hasse diagrams of the lattices for (a) the concrete domain $(2^{\mathbb{Z}_n}, \subseteq)$ and (b) the abstract domain $(\mathbb{T}_n, \sqsubseteq_{\mathbb{A}})$ for $n = 2$. Below every element of the abstract domain, we show its Linux kernel implementation using two 2-bit values (v, m). Also shown are two examples of abstraction (α, dotted black lines) followed by concretization (γ, solid black lines). (i) Starting with $C' = \{1, 2, 3\}$, $\alpha(C')$ gives $\mu\mu$, and $\gamma(\alpha(C'))$ gives $\{0, 1, 2, 3\}$, an overapproximation of C'. (ii) However, starting with $C'' = \{2, 3\}$, $\alpha(C'')$ gives 1μ, and $\gamma(\alpha(C''))$ gives $\{2, 3\}$, exactly equal to C''. In both cases, $C \sqsubseteq_{\mathbb{C}} \gamma(\alpha(C))$.

t_i is a trit with value 0, 1, or μ. Any of 3^n abstract values can be used to represent concrete sets of n-bit values.

Implementation of tnums in the Linux kernel. The Linux kernel's implementation of representing one n-trit tnum $P \in \mathbb{T}_n$ uses two n-bit values $(P.v, P.m)$, where the 'v' stands for *value* and the 'm' stands for *mask.* The values of the kth bits of $P.v$ and $P.m$ are used to inform the value of the kth trit of P.

$$\begin{aligned} (P.v[k] = 0 \wedge P.m[k] = 0) &\triangleq P[k] = 0 \\ (P.v[k] = 1 \wedge P.m[k] = 0) &\triangleq P[k] = 1 \\ (P.v[k] = 0 \wedge P.m[k] = 1) &\triangleq P[k] = \mu \end{aligned} \tag{3}$$

We define the domain of abstract values $\mathbb{T}_n \triangleq \mathbb{Z}_n \times \mathbb{Z}_n$. If for a tnum P, $P.v[k] = P.m[k] = 1$ at some position k, we say that such a tnum is not *well-formed*. All such tnums represent the abstract value $\bot$ and the concrete empty set $\varnothing$.

$$\forall P : (\exists k : \;\; P.v[k] \;\texttt{\&}\; P.m[k] = 1) \Leftrightarrow \;\; P = \bot \tag{4}$$

A large fraction of random bit patterns (v, m) aren't well formed: in particular, only 3^n among the 2^{2n} n-bit (v, m) bit patterns correspond to well-formed tnums that are not $\bot$.

We are now ready to define the Galois connection for the tnum abstract domain using the above implementation of abstract values. These take a form similar to the functions defined in prior work [41]. In the discussion that follows we will use the notation (&, |, ⊕, ~, <<, >>) respectively for the bitwise and, or, exclusive-or, negation, left-shift, and right-shift operations over n-bit bit vectors.

Galois connection. Given a concrete set $C \in 2^{\mathbb{Z}_n}$. The abstraction function $\alpha : 2^{\mathbb{Z}_n} \rightarrow \mathbb{Z}_n \times \mathbb{Z}_n$ is defined as follows.

$$\begin{aligned} \alpha_{\texttt{\&}}(C) &\triangleq \texttt{\&}\{c \mid c \in C\} \\ \alpha_{\texttt{|}}(C) &\triangleq \texttt{|}\{c \mid c \in C\} \\ \alpha(C) &\triangleq (\; \alpha_{\texttt{\&}}(C), \;\; \alpha_{\texttt{\&}}(C) \oplus \alpha_{\texttt{|}}(C) \;) \end{aligned} \tag{5}$$

This abstraction function is sound. However, it is not exact, as easily seen from the fact that there are 2^{2^n} elements in $\mathbb{C}$ but only 3^n well-formed tnums in $\mathbb{T}_n$. Many concrete sets will be over-approximated. However, α is a composition of functions that abstract the domain exactly *when each bit is considered separately* [48, 3]. Informally, given a concrete set $C \in \mathbb{C}$ and $x, y \in C$, $\alpha(C)$ contains an uncertain trit at position k iff C contains x and y with bits differing at k.

$$\begin{aligned} \forall b \in \{0, 1\} : \;\; \alpha(C)[k] = b \;\; &\Leftrightarrow \;\; \forall x \in C : \;\; x[k] = b \\ \alpha(C)[k] = \mu \;\; &\Leftrightarrow \;\; \exists x, y \in C : \;\; x[k] = 0 \wedge y[k] = 1 \end{aligned} \tag{6}$$

This abstraction function α is *bitwise exact*.

Further, consider a tnum $P \in \mathbb{T}_n$ implemented as $(P.v, P.m) \in \mathbb{Z}_n \times \mathbb{Z}_n$. Then the concretization function $\gamma : \mathbb{Z}_n \times \mathbb{Z}_n \rightarrow 2^{\mathbb{Z}_n}$ is defined as:

$$\begin{aligned} \gamma(P) \;=\; \gamma((P.v, P.m)) &\triangleq \{c \in \mathbb{Z} \mid c \;\texttt{\&}\; \texttt{\textasciitilde} P.m = P.v\} \\ \gamma(\bot) &\triangleq \varnothing \end{aligned} \tag{7}$$

Then α and γ form a Galois connection. Informally, the tnum obtained from applying α on a set of concrete values always soundly over-approximates the original set if concretized. An illustration of this fact can be seen in Fig. 1. Please refer to the extended technical report [49] for the (standard) proof. The existence of the Galois connection enables, in principle, constructing sound and optimal abstract operators over tnums. The abstraction of the concrete set $\{1, 2, 3\}$ soundly overapproximates it: $\{1, 2, 3\} \sqsubseteq_{\mathbb{C}} \gamma(\alpha(\{1, 2, 3\}))$.

Abstract operators on tnums. The BPF instruction set supports the following (typical) concrete operations over 64-bit registers: `add, sub, mul, div, or, and, lsh, rsh, neg, mod, xor` and `arsh`. To soundly analyze general BPF programs, the BPF static analyzer requires abstract operators corresponding to all the above concrete operations. For some operators, notably `div` and `mod`, defining a precise abstract operator is challenging. In such cases, the BPF static analyzer conservatively and soundly sets all the output trits to unknown.

Challenges. Despite enjoying a Galois connection, constructing *efficient* optimal abstractions for arithmetic operators is non-trivial. Given a concrete operator f, the optimal abstract operator $\alpha \circ f \circ \gamma$ is infeasible to compute in practice [50, 51, 52]. For example, if f is a concrete operator of arity 2, there may be 2^{2n} computations of f after the first concretization $\gamma(.)$ in the worst case (the average case is not much better).

Prior work on the bitfield domain [41], a domain similar to tnums $(2^{\mathbb{Z}_n} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathbb{Z}_n \times \mathbb{Z}_n)$, presents abstract operators for bitwise or, and, exclusive-or, left and right shift operations that are *optimal*. However, most prior works on the bitfield and known bits abstract domains [43, 3, 44, 40, 41] fail to provide

abstract arithmetic operators for addition, subtraction, and multiplication. To our knowledge, Regehr and Duongsaa [42] provide the only known abstract operators for arithmetic in this domain, based on ripple-carry logic and composition of abstract operators. These operators are sound but not optimal. Further, they have a runtime of $O(n)$ for n-bit abstract addition and subtraction, and $O(n^2)$ for abstract multiplication.

In the next section, we present proofs of soundness and optimality for abstract operators for addition and subtraction originally developed (without formal proof) in the Linux kernel. These operators run in $O(1)$ time given n-bit machine arithmetic instructions ($n = 64$ in the kernel). Such efficiency is remarkable, given that in general addition and subtraction use ripple-carry operations creating dependencies between the bits. We also present an abstract multiplication operator that is provably sound, empirically more precise, and faster than the abstract multiplication in [42] and the Linux kernel. Notably, none of the algorithms in this paper use the composition structure $\alpha \circ f \circ \gamma$ or "merely" compose existing sound abstract operators. This motivated us to develop dedicated proof techniques.

III. Soundness and Optimality of Abstract Arithmetic over Tnums

We explore the soundness and optimality of tnum arithmetic operators, specifically addition, subtraction, and multiplication. The kernel proposes abstract operators for each of them, but lacks any proof of soundness. Hence, we perform an automated (bounded bitwidth) verification of the soundness of the kernel's tnum abstract operators (§III-A) using SMT solvers. We were able to prove the soundness of the kernel's abstract addition, subtraction, and all other bitwise operators up to 64-bits, and soundness of the kernel's multiplication up to 8-bits. Motivated by these results, we undertook an analytical study of these algorithms, which led us to paper-and-pen proofs of both *soundness and optimality* of the kernel's abstract operators for addition and subtraction over *unbounded bitwidths* (§III-B). We were unable to analytically prove the soundness of the kernel's tnum multiplication for unbounded bitwidths. Hence, we developed a new algorithm for tnum multiplication that is provably sound for unbounded bitwidths, and empirically more precise and faster than all prior implementations (§III-C).

A. Automatic Bounded Verification of Kernel Tnum Arithmetic

We encode verification conditions corresponding to the soundness of tnum abstract arithmetic operators in first order logic and discharge them to a solver. We use the theory of bitvectors. Our verification conditions are specific to a particular bitwidth (n). We use 64-bit bitvectors to encode the tnum operations wherever feasible ($n = 64$ in the kernel). For a tnum P drawn from the set of n-trit tnums $\mathbb{T}_n$, we denote its kernel implementation by $(P.v, P.m) \in \mathbb{Z}_n \times \mathbb{Z}_n$.

Soundness of 2-ary operators. Recall from Section §II the notion of soundness of an abstract operator. We can generalize this notion to 2-ary operators $\mathtt{op}_{\mathbb{T}} : \mathbb{T}_n \times \mathbb{T}_n \to \mathbb{T}_n$ and $\mathtt{op}_{\mathbb{C}} : \mathbb{Z}_n \times \mathbb{Z}_n \to \mathbb{Z}_n$. We say that $\mathtt{op}_{\mathbb{T}}$ is a sound abstraction of $\mathtt{op}_{\mathbb{C}}$ iff the following condition (Eqn. 8) holds.

$$\begin{aligned}&\forall P, Q \in \mathbb{T}_n : \\ &\Big\{\mathtt{op}_{\mathbb{C}}(x, y) \mid x \in \gamma(P), y \in \gamma(Q)\Big\} \sqsubseteq_{\mathbb{C}} \gamma(\mathtt{op}_{\mathbb{T}}(P, Q))\end{aligned} \tag{8}$$

To encode (8) in first-order logic, recall that the concrete order $\sqsubseteq_{\mathbb{C}}$ is just the subset relationship between the two sets $\subseteq$. At a high level, the subset relationship $S_1 \subseteq S_2$ in (8) can be encoded by universally quantifying over the members of S_1 and writing down the query $\forall x \in \mathbb{Z}_n : x \in S_1 \Rightarrow x \in S_2$. The formula $x \in S_1$ is easy to encode given the left-hand side of (8). To encode $x \in S_2$ from the right-hand side of (8), we define a *membership predicate*. This predicate asserts that $x \in \gamma(R)$ where $R \triangleq \mathtt{op}_{\mathbb{T}}(P, Q)$. Finally, we ensure that the universally quantified tnums P and Q are non-empty, and encode the action of the concrete and abstract operators $\mathtt{op}_{\mathbb{C}}$ and $\mathtt{op}_{\mathbb{T}}$ in logic. The details follow.

Membership predicate $x \in \gamma(P)$. Consider a concrete value x that is contained in the concretization of tnum P. Using the definition of the concretization function in (7), we write down the predicate $member$:

$$member(x, P) \triangleq x \,\&\, \mathord{\sim} P.m = P.v \tag{9}$$

Quantifying over well-formed tnums. To ensure that (8) only quantifies over non-empty tnums, we encode one more predicate, $wellformed$, based on (4):

$$wellformed(P) \triangleq P.v \,\&\, P.m = 0 \tag{10}$$

Putting it all together. The soundness predicate for a given pair of abstract and concrete operators $\mathtt{op}_{\mathbb{T}}, \mathtt{op}_{\mathbb{C}}$ is

$$\begin{aligned}&\forall P, Q \in \mathbb{T}_n, x, y \in \mathbb{Z}_n : \\ &\quad wellformed(P) \wedge wellformed(Q) \wedge member(x, P) \\ &\quad \wedge member(y, Q) \wedge z = \mathtt{op}_{\mathbb{C}}(x, y) \wedge R = \mathtt{op}_{\mathbb{T}}(P, Q) \\ &\quad \Rightarrow member(z, R)\end{aligned} \tag{11}$$

An SMT solver can show the validity of this formula by proving that the negation of this formula is unsatisfiable.

Example: encoding abstract tnum addition. We show how to encode the soundness of the abstract addition operator over tnums. The kernel uses the algorithm `tnum_add` from Listing 1 to perform abstract addition over two tnums. The predicate add below captures the result of abstract addition of P and Q into R.

$$\begin{aligned}&add(P, Q, R) \triangleq \\ &\quad (sv = P.v + Q.v) \wedge (sm = P.m + Q.m) \wedge (\Sigma = sv + sm) \\ &\quad \wedge (\chi = \Sigma \oplus sv) \wedge (\eta = \chi \mid P.m \mid Q.m) \wedge (R.v = sv \,\&\, \mathord{\sim} \eta) \\ &\quad \wedge (R.m = \eta)\end{aligned} \tag{12}$$

We can plug in the add predicate in place of $\mathtt{op}_{\mathbb{T}}$ in Eqn. 11. The function $\mathtt{op}_{\mathbb{C}}$ is just n-bit bitvector addition.

```
1  def tnum_add(tnum P, tnum Q):
2
3     u64 sv := P.v + Q.v
4     u64 sm := P.m + Q.m
5     u64 Σ := sv + sm
6     u64 χ := Σ ⊕ sv
7     u64 η := χ | P.m | Q.m
8     tnum R := tnum(s_v & ~η, η)
9     return R
```

Listing 1: Linux kernel's implementation of tnum addition (`tnum_add`)

```
1  def  kern_mul(tnum P, tnum Q)
2
3     tnum π := tnum(P.v * Q.v, 0)
4     tnum ACC := hma(π, P.m, Q.m | Q.v)
5     tnum R:= hma(ACC, Q.m, P.v)
6     return R
7
8  def hma(tnum ACC, u64 x, u64 y)
9
10    while (y):
11       if (y[0] == 1)
12          ACC := tnum_add(ACC, tnum(0, x))
13       y := y >> 1
14       x := x << 1
15    return ACC
```

Listing 2: Linux kernel's implementation of tnum multiplication (`kern_mul`)

Observations from bounded verification. We encoded the first-order logic formulas to perform bounded verification of the soundness of the following tnum operators defined in the Linux kernel: addition, subtraction, multiplication, bitwise or, bitwise and, bitwise exclusive-or, left-shift, right-shift, and arithmetic right-shift. We have spot-checked the correctness of our encodings with respect to the kernel source code using randomly-drawn tnum inputs; the details of this testing harness are in our extended technical report [49].

For all operators except multiplication, verification succeeded for bitvectors of width 64 in just a few seconds. In contrast, verification of multiplication (`kern_mul`), shown in Listing 2, succeeds quickly at bitwidth $n = 8$, but does not complete even after 24 hours with bitwidth $n = 16$. This is due to the presence of non-linear operations and large unrolled loops. This observation motivated us to develop a new, provably sound algorithm for tnum multiplication (§III-C).

Further, our bounded verification efforts helped us uncover non-obvious properties of tnum arithmetic: (1) tnum addition is not associative, (2) tnum addition and subtraction are not inverse operations, and (3) tnum multiplication is not commutative.

B. Soundness and Optimality of Tnum Abstract Addition

We present an analytical proof of the soundness and optimality of the kernel's abstract addition operator for unbounded bitwidths. The proof for subtraction, which is very similar in structure, is in our extended technical report [49].

An example. The source code for abstract addition (`tnum_add`) is shown in Listing 1. Figure 2 illustrates tnum addition with an

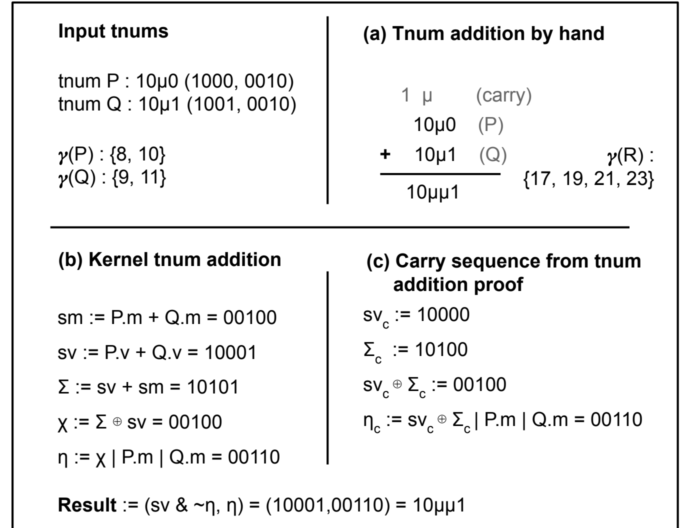

Fig. 2: Illustration of tnum addition. We provide a side by side comparison of (a) tnum addition by hand and (b) the kernel algorithm for tnum addition as well as (c) the carry sequence in the operation as discussed in the proof of tnum addition.

example. In particular, adding two tnums "by hand", as shown in Fig. 2(a), propagates uncertainty explicitly in the carries, rippling the carry bits through the tnums one bit position at a time. However, as seen in Fig. 2(b), `tnum_add` does not use any such ripple-carry structure in its computations. Yet, as we show later (and illustrated in Fig. 2(c)), `tnum_add` implicitly reasons about the unknown bits in the sequence of carries produced during the addition.

***Definition* 1. Full adder equations.** When adding two concrete binary numbers p and q, each bit of the addition result r is set according to the following:

$$r[i] = p[i] \oplus q[i] \oplus c_{in}[i]$$

where $\oplus$ is the exclusive-or operation and $c_{in}[i] = c_{out}[i-1]$ and $c_{out}[i-1]$ is the carry-out from the addition in bit position $i-1$. The carry-out bit at the ith position is given by

$$c_{out}[i] = (p[i] \,\&\, q[i]) \mid (c_{in}[i] \,\&\, (p[i] \oplus q[i]))$$

Key proof technique. We show the soundness and optimality of `tnum_add` by reasoning about the set of all possible *concrete outputs*, *i.e.,* the results of executions of concrete additions over elements of the input tnums $P, Q \in \mathbb{T}$. If we denote by $+$ the concrete addition operator over $\mathbb{Z}_n$, this is the set $\{p + q \mid p \in \gamma(P) \wedge q \in \gamma(Q)\}$ or $+(\gamma(P), \gamma(Q))$ in short. The proof proceeds by finding bit positions in the concrete output set that can be shown to be either a 1 or a 0 in *all members* of that set (respectively lemmas 2 and 3). Every other bit position is such that there are elements in the concrete output set that differ at that bit position. Lemma 4 invokes the bitwise-exactness (Eqn. 6) of the abstraction function α, and along with Lemma 5, shows that `tnum_add` is a sound and optimal abstraction for $+$ (*i.e.,* the same as $\alpha \circ + \circ \gamma$).

Consider the addition that occurs "by hand" in Fig. 2(a). Intuitively, at a given bit position of the output tnum, the result

will be unknown if either of the operand bits $p[i]$ or $q[i]$ is unknown, or if the carry-in bit $c_{in}[i]$ (generated from less-significant bit positions) is unknown. Note that these three bits may be (un)known independent of each other since they depend on different parts of the input tnums. The crux of the proof lies in identifying which carry-in bit positions vary across different concrete additions. This is done by distinguishing the carries generated due to the unknown bits in the operands from the carries that will be present or absent in *any* concrete addition drawn from the input tnums. In the example in Fig. 2(a), the sequence of carries is $10\mu00$, with the middle carry-in bit being uncertain and all others known to be 0s or 1s in all concrete additions from the input tnums.

Suppose p and q are two concrete values in tnum P and tnum Q, respectively, *i.e.,* $p \in \gamma(P), q \in \gamma(Q)$.

Lemma 2. **Minimum carries lemma.** The addition $sv = P.v + Q.v$ will produce a sequence of carry bits that has the least number of 1s out of all possible additions $p + q$.

The consequence of this lemma is that *any* concrete addition $p + q$ will produce a sequence of carry bits with 1s in at least those positions where the sv addition produced carry bits set to 1 (the extended technical report provides a proof of this lemma). Fig. 2(c) shows the set of carries produced in sv (*i.e.,* $sv_c \triangleq 10000$). Any addition $p + q$ will produce a 1-bit carry in the same positions as the 1 bits in sv_c.

Lemma 3. **Maximum carries lemma.** The addition $\Sigma = (P.v + P.m) + (Q.v + Q.m)$ will produce the sequence of carry bits with the most number of 1s out of all possible additions $p + q$.

The consequence of this lemma is that *any* concrete addition $p + q$ will produce a sequence of carry bits with 0s in at least those positions where the Σ addition produced carry bits set to 0 (proof is available in our extended technical report). Fig. 2(c) shows the set of carries produced in Σ (*i.e.,* $\Sigma_c \triangleq 10100$). Any addition $p + q$ will produce a 0-bit carry in the same positions as the 0 bits in Σ_c.

Lemma 4. **Capture uncertainty lemma.** Let sv_c and Σ_c be the sequence of carry-in bits from the additions in sv and Σ, respectively. Suppose $\chi_c \triangleq sv_c \oplus \Sigma_c$. The bit positions k where $\chi_c[k] = 0$ have carry bits fixed in all concrete additions $p + q$ from $+(\gamma(P), \gamma(Q))$. The bit positions k where $\chi_c[k] = 1$ vary depending on the concrete addition: *i.e.,* $\exists p_1, p_2 \in \gamma(P), q_1, q_2 \in \gamma(Q)$ such that $p_1 + q_1$ has its carry bit set at position k but $p_2 + q_2$ has that bit unset.

Intuitively, from the minimum carries lemma, any carry bit that is set in sv_c must be set in the sequence of carry bits in any concrete addition $p + q$. Similarly, from the maximum carries lemma, any carry bit that is unset in Σ_c must be unset in the sequence of carry bits in any concrete addition $p + q$. Hence, $sv_c \oplus \Sigma_c$ represents the carries that may arise purely from the uncertainty in the concrete operands picked from P and Q. Further, these carries do in fact differ in two concrete additions sv and Σ. From the bitwise-exactness of the tnum abstraction function α (Eqn. 6), it follows that these are precisely the bits that must be unknown in the resulting tnum due to the carries. See the extended technical report for a detailed proof.

Hence, the mask in the resulting tnum must be $(sv_c \oplus \Sigma_c) \mid P.m \mid Q.m$. However, `tnum_add` uses the final mask $(sv \oplus \Sigma) \mid P.m \mid Q.m$ (see Listing 1). Lemma 5 shows that these two quantities are, in fact, always the same.

Lemma 5. **Equivalence of mask expressions.** The expressions $(sv \oplus \Sigma) \mid P.m \mid Q.m$ and $(sv_c \oplus \Sigma_c) \mid P.m \mid Q.m$ compute the same result.

We prove this lemma using the rules of propositional logic in our extended technical report. Together, these lemmas allow us to show the soundness and optimality of `tnum_add` below.

Theorem 6. **Soundness and optimality of tnum_add** The algorithm `tnum_add` shown in Listing 1 is a sound and optimal abstraction of concrete addition over n-bit bitvectors for unbounded n.

C. Sound and Efficient Tnum Abstract Multiplication

This section describes a novel algorithm for tnum multiplication and a proof that it is a sound abstraction of multiplication of n-bit concrete values for unbounded n. Our algorithm has $O(n)$ run time. It is not an optimal abstraction of concrete multiplication. However, as we show later (§IV), our algorithm is empirically more precise and faster than all known prior implementations of multiplication in this abstract domain. We were able to contribute our algorithm to the tnum implementation in the latest Linux kernel.

```
1  def our_mul_simplified(tnum P, tnum Q):
2
3     ACCv := tnum(0, 0)
4     ACCm := tnum(0, 0)
5
6     # loop runs bitwidth times
7     for i in range(0, bitwidth):
8        # LSB of tnum P is a certain 1
9        if (P.v[0] == 1) and (P.m[0] == 0):
10         ACCv := tnum_add(ACCv, tnum(Q.v, 0))
11         ACCm := tnum_add(ACCm, tnum(0, Q.m))
12       # LSB of tnum P is uncertain
13       else if  (P.m[0] == 1):
14         ACCm := tnum_add(ACCm, tnum(0, Q.v|Q.m))
15       # Note: no case for LSB is certain 0
16       P := tnum_rshift(P, 1)
17       Q := tnum_lshift(Q, 1)
18
19    tnum R := tnum_add(ACCv, ACCm)
20    return R
```

Listing 3: A simplified implementation of our tnum multiplication algorithm (`our_mul_simplified`).

Our algorithm `our_mul` ***through an example.*** Our tnum abstract multiplication algorithm is shown in Listing 4. The algorithm in Listing 3 is semantically equivalent to it, but easier to understand, so we explain the algorithm and its proof primarily using the algorithm in Listing 3.

Similar to the prior multiplication algorithms proposed in bit-level reasoning domains [42, 39], our algorithm is inspired

by the long multiplication method to generate the product of two binary values. The algorithm proceeds in a single loop iterating over the bitwidth of the input tnums. `Our_mul` takes two input tnums P and Q, and returns a result R.

Fig. 3(a) shows an example. Suppose we are given tnums $P = \mu 01$ ($P.v = 001, P.m = 100$) and $Q = \mu 10$ ($Q.v = 010, Q.m = 100$) to multiply. Two fully concrete n-bit binary numbers may be multiplied in two steps: (i) by computing the products of each bit in the multiplier (P) with the multiplicand (Q), to generate n *partial products,* and (ii) adding the n partial products after appropriately bit-shifting them. To generalize long multiplication to n-bit tnums which contain unknown (μ) trits, we add new rules: $0 * \mu = 0; 1 * \mu = \mu$; and $\mu * \mu = \mu$. Since the partial products themselves contain unknown trits, the addition of the partial products must occur through the abstract addition operator `tnum_add`.

```
1  def  our_mul(tnum P, tnum Q):
2
3     ACCv := tnum(P.v * Q.v, 0)
4     ACCm := tnum(0, 0)
5
6     while P.value or P.mask:
7        # LSB of tnum P is a certain 1
8        if (P.v[0] == 1) and (P.m[0] == 0):
9          ACCm := tnum_add(ACCm, tnum(0, Q.m))
10       # LSB of tnum P is uncertain
11       else if  (P.m[0] == 1):
12         ACCm := tnum_add(ACCm, tnum(0, Q.v|Q.m))
13       # Note: no case for LSB is certain 0
14       P := tnum_rshift(P, 1)
15       Q := tnum_lshift(Q, 1)
16
17    tnum R := tnum_add(ACCv, ACCm)
18    return R
```

Listing 4: Our final tnum multiplication algorithm (`our_mul`).

Our tnum multiplication algorithm for the same pair of inputs is illustrated in Fig. 3(b). The algorithm uses two tnums, ACC_V and ACC_M, which are initialized $(v, m) \triangleq (0, 0)$. The tnums ACC_V and ACC_M accumulate abstract partial products generated in each iteration using tnum abstract additions (`tnum_add`). The algorithm proceeds as follows:

1) If the least significant bit of any concrete value $x \in \gamma(P)$ is known to be 1, then ACC_V (resp. ACC_M) accumulates the known bits (resp. unknown bits) in Q (*e.g.,* iteration 1);
2) If the least significant bit of any $x \in \gamma(P)$ is known to be 0, ACC_V and ACC_M remain unchanged (*e.g.,* iteration 2);
3) If the least significant bit of P is unknown (μ), then ACC_V is unchanged, but ACC_M accumulates a tnum with a mask such that all possible bits that may be set in any $x \in \gamma(P)$ are also set in the mask (*e.g.,* iteration 3).

At the end of each iteration, P (resp. Q) is bit-shifted to the right (resp. left) by 1 position to ensure that the next partial product is appropriately shifted before addition. The specific methods of updating ACC_V and ACC_M in each iteration make `our_mul` distinct from prior multiplication algorithms [42, 39]. In particular, `our_mul` decomposes the accumulation of partial

Input tnums
tnum P : μ01 (010, 100)
tnum Q : μ10 (001, 100)
γ(P) : {1, 5}
γ(Q) : {2, 6}

(a) Tnum multiplication by hand
μ10 (Q)
x μ01 (P)
μ10
0000
μμ000
μμμ10 (R)
γ(R) : {2, 6, 10, 14, 18, 22, 26, 30}

(b) Our tnum multiplication algorithm (* denotes unchanged value)

Iteration 1	Iteration 2	Iteration 3
(p_0 is certain 1)	(p_0 is certain 0)	(p_0 is uncertain)
ACC_V := (00010, 0)	ACC_V := (00010, 0) *	ACC_V := (00010, 0) *
ACC_M := (0,00100)	ACC_M := (0, 00100) *	ACC_M := (0, 11100)
P := 00μ0	P := 0000μ	P := 000000
Q := μ100	Q := μ1000	Q := μ10000
μ10	μ10 + 0000	μ10 + 0000 + μμ000

Final result
Result := (ACC_V + ACC_M) = (00010, 11100) = μμμ10

Fig. 3: Illustration of our tnum multiplication. We provide a side by side comparison of (a) tnum multiplication by hand and (b) our improved algorithm for tnum multiplication.

products into two tnums and uses just a single loop over the bitwidth. These modifications are crucial to the precision and efficiency of `our_mul` (§IV).

Key proof techniques. Recall that a (unary) abstract operator g is a sound abstraction of a concrete operator f if $\forall a \in \mathbb{A} : f(\gamma(a)) \sqsubseteq_{\mathbb{C}} \gamma(g(a))$. We show that our abstract multiplication algorithm `our_mul` is sound by showing that $\{x * y \mid x \in \gamma(P) \wedge y \in \gamma(Q)\} \sqsubseteq_{\mathbb{C}} \gamma(\texttt{our_mul}(P, Q))$ for any tnums $P, Q \in \mathbb{T}_n$. We denote the former set $*(\gamma(P), \gamma(Q))$ in short. The $*$ is the concrete multiplication over n-bit bitvectors.

All the known abstract multiplication algorithms in this domain are composed of abstract additions and abstract shifts. A typical approach to prove soundness of such operators is to invoke the result that when sound abstract operators are composed soundly, *i.e.,* in the same way as the corresponding concrete operators are composed, the result is a sound abstraction of the composed concrete operator [45, Theorem 2.6]. The soundness of the abstract multiplication from Regehr and Duongsaa [42] may be proved as a special case of this general result. However, this approach is not applicable to proving the soundness of `our_mul`, since `our_mul`'s composition does not mirror any composition of (concrete) additions and shifts to produce a product. Instead, we are forced to develop a proof specifically for `our_mul` by observing, through two intermediate lemmas (Lemma 9 and Lemma 8) that the concrete products in $*(\gamma(P), \gamma(Q)) \in \gamma(\texttt{our_mul}(P, Q))$.

Below, we show a sketch of the proof of the soundness of `our_mul_simplified`, and argue (Lemma 11) that `our_mul` is equivalent to `our_mul_simplified`.

Observation 7. For two concrete bitvectors x and y of width n bits, the result of multiplication $y * x$ is just

$$y * x = \sum_{k=0}^{n-1} x[k] * (y \texttt{ << } k)$$

We call each term $x[k] * (y \ll k)$ a *partial product.*

Lemma 8. **Tnum set union with zero.** Given a non-empty tnum $P \in \mathbb{Z}_n \times \mathbb{Z}_n$, define $Q \triangleq tnum(0, P.v \mid P.m)$. Then, (i) $x \in \gamma(P) \Rightarrow x \in \gamma(Q)$, and (ii) $0 \in \gamma(Q)$.

Intuitively, any tnum $(0, m)$ when concretized contains the value 0. Further, building a new tnum Q whose mask has set all the bits corresponding to the set value or mask bits of P ensures that $\gamma(P) \sqsubseteq_{\mathbb{C}} \gamma(Q)$. The full proof is in our extended technical report [49]. For example, given $P = 0\mu1 = (001, 010)$ and $Q \triangleq (0, 011)$, we have $\gamma(P) \sqsubseteq_{\mathbb{C}} \gamma(Q)$ and $0 \in \gamma(Q)$.

For the next lemma, we define a variable-arity version of `tnum_add` as follows: $\texttt{tnum_add}_{j=0}^{n-1}(T_j)$ is evaluated by folding the `tnum_add` operator over the list of tnum operands $T_0, T_1, \cdots, T_{n-1}$ from left to right.

Lemma 9. **Value-mask-decomposed tnum summations.** Given n non-empty tnums $T_0, T_1, \ldots, T_{n-1} \in \mathbb{T}_n$. Suppose we pick n values $z_0, z_1, z_2, \ldots, z_{n-1} \in \mathbb{Z}_n$ such that $\forall\, 0 \leq j \leq n-1 : z_j \in \gamma(T_j)$. Define tnum

$$S \triangleq \texttt{tnum_add}(\texttt{tnum_add}_{j=0}^{n-1} \ (tnum(T_j.v, 0)), \texttt{tnum_add}_{j=0}^{n-1} \ (tnum(0, T_j.m)))$$

where $\texttt{tnum_add}_{(\cdot)}^{(\cdot)}$ is a variable-arity version of `tnum_add` defined above. Then, $\sum_{j=0}^{n-1} z_j \in \gamma(S)$.

Intuitively, suppose we had n tnums $T_i, 0 \leq i \leq n-1$ and we seek to construct a new tnum S whose concretization $\gamma(S)$ contains all possible *concrete sums* from the T_i, *i.e.,* such that $\{\sum_{j=0}^{n-1} x_j \mid x_i \in \gamma(T_i)\} \sqsubseteq_{\mathbb{C}} \gamma(S)$. The most natural method to construct such a tnum S is to use the sound abstract addition operator `tnum_add` over the T_i, *i.e.,* $\texttt{tnum_add}_{j=0}^{n-1}(T_j)$. This lemma provides another method of constructing such a tnum S: decompose the tnums T_i each into two tnums, consisting of the values and the masks separately. Use `tnum_add` to separately add the value tnums, add the mask tnums, and finally add the two resulting tnums from the value-sum and mask-sum to produce S. Then S contains all concrete sums. The full proof of this lemma is in the extended technical report. For example, suppose $T_1 = 1\mu0 = (100, 010), T_2 = 01\mu = (010, 001)$. Then $\forall x_1 \in \gamma(T_1), x_2 \in \gamma(T_2) : x_1 + x_2 \in \gamma(\texttt{tnum_add}((110, 0), (0, 011)))$.

Theorem 10. **Soundness of `our_mul`.** $\forall\, x \in \gamma(P),\ y \in \gamma(Q)$ the result R returned by `our_mul_simplified` (Listing 3) is such that $x * y \in \gamma(R)$, assuming that abstract tnum addition (`tnum_add`) and abstract tnum shifts (`tnum_lshift`, `tnum_rshift`) are sound.

We prove this theorem by showing three properties, whose full proofs are in the extended technical report. Below, P_{in} and Q_{in} are the formal parameters to `our_mul`.

Property P1. P **and** Q **are bit-shifted versions of** P_{in} **and** Q_{in}**.** This property follows naturally from the algorithm, which only updates the tnums P and Q in the code using tnum bit-shift operations (`tnum_lshift`, `tnum_rshift`).

Property P2. ACC_V **and** ACC_M **are value-mask-decomposed summations of partial products.** There exist tnums $T_0, T_1, \ldots, T_{n-1}$ such that (i) any concrete j^{th} partial product, $z_j \triangleq x[j] * (y \ll j) \in \gamma(T_j)$, for $0 \leq j \leq n-1$; (ii) at the end of the kth iteration of the loop, $ACC_V = \texttt{tnum_add}_{j=0}^{k-1}(tnum(T_j.v, 0))$, and (iii) at the end of the kth iteration of the loop, $ACC_M = \texttt{tnum_add}_{j=0}^{k-1}(tnum(0, T_j.m))$.

At a high level, this property states that there is a set of tnums T_j, where $\gamma(T_j)$ contains all possible concrete values of the jth partial product term $z_j \triangleq x[j] * (y \ll j)$ (Observation 7). In the example in Fig. 3(b), $T_0 = \mu10, T_1 = 0000, T_2 = \mu\mu000$. In the case where $P[0]$ is μ, we use Lemma 8 to show that the T_j constructed by `our_mul_simplified` is such that $\gamma(Q) \sqsubseteq_{\mathbb{C}} \gamma(T_j)$ and $0 \in \gamma(T_j)$. We also show that ACC_V is the value-sum of the T_j (see Lemma 9) while ACC_M is the mask-sum. In Fig. 3(b), $ACC_V \triangleq \texttt{tnum_add}(010, 0000, 00000)$ and $ACC_M \triangleq \texttt{tnum_add}(\mu00, 0000, \mu\mu000)$.

Property P3. **(Product containment)** $\sum_{j=0}^{n-1} z_j \in \gamma(\texttt{tnum_add}(ACC_V, ACC_M))$. That is, $\forall x \in \gamma(P), y \in \gamma(Q) : x * y \in \gamma(\texttt{tnum_add}(ACC_V, ACC_M))$.

This result follows from property P2 and Lemma 9. Property P3 concludes a proof of soundness of `our_mul_simplified`: $\forall x \in \gamma(P), y \in \gamma(Q) : x * y \in \gamma(R)$.

Lemma 11. **Correctness of strength reductions.** `Our_mul` (Listing 4) is equivalent to `our_mul_simplified` (Listing 3) in terms of its input-output behavior.

The existence of two accumulating tnums ACC_V and ACC_M in `our_mul_simplified` allows us to use Lemma 9 to prove soundness. However, it is unnecessary to construct ACC_V iteration by iteration. We observe that ACC_V is merely accumulating $(Q.v, 0)$ whenever $P[0]$ is known to be 1. All bits in each tnum accumulated into ACC_V are known. When `tnum_add` is used to add n tnums $(v_i, 0), 0 \leq i \leq n-1$ it is easy to see that the result is $(\sum_{i=0}^{n-1} v_i, 0)$. Using Observation 7, we see that at end of the loop, $ACC_V = tnum(P.v * Q.v, 0)$. As an added optimization, `our_mul` soundly terminates the loop early if $P = (0, 0)$ at the beginning of any iteration. Since `our_mul` and `our_mul_simplified` are equivalent, `our_mul` is also a sound abstraction of $*$.

While `our_mul` is sound, it is not optimal. Key questions remain in designing a sound and optimal algorithm with $O(n)$ or faster run time. (1) How can we incorporate correlation in unknown bits across partial products? For example, multiplying $P = 11, Q = \mu1$ produces the partial products $T_1 = 11, T_2 = \mu\mu0$. However, the two μ trits in T_2 are concretely either both 0 or both 1, resulting from the same μ trit in Q. Failing to consider this in the addition makes the result imprecise. (2) Can we design a sound, precise, and fast tnum addition operator of arity n? (3) Eschewing long multiplication, is it possible to use concrete multiplication $(*)$ over tnum masks to determine the unknown bits in the result?

IV. Experimental Evaluation

Tnum operations are only one component of the Linux kernel's BPF analyzer. To keep our measurement and contributions focused, our evaluation focuses on the precision and speed of our tnum multiplication operation relative to prior algorithms.

Prior algorithms for abstract multiplication. Apart from Linux kernel's multiplication [39], Regehr and Duongsaa [42] also provide an algorithm for abstract multiplication in a domain similar to tnums, which is called the *bitwise* domain. Listing 5 presents their multiplication algorithm, which we call `bitwise_mul`. We experimentally compare the precision and performance of our tnum multiplication with both `bitwise_mul` and the Linux kernel version (`kern_mul`). We have performed bounded verification of the soundness of both `kern_mul` and `bitwise_mul` at bitwidth $n = 8$.

Similar to `our_mul`, `bitwise_mul` is also inspired from long multiplication. It generates partial products that are subsequently added after appropriately bit-shifting them. We observed that `bitwise_mul` needs to be carefully implemented with machine arithmetic operations to have reasonable performance. In `bitwise_mul`, the function `multiply_bit` modifies the second operand Q based on the ith trit of the first operand P. If the ith trit of P is μ, this modification is done in a trit-by-trit fashion (*i.e.,* by iterating over the n trits of Q and setting them to μ). To improve `bitwise_mul`'s performance with tnums, we substituted this loop with a single tnum operation in our implementation: when P is μ, we construct a new tnum `(0, Q.mask | Q.value)`, which has the same effect as individually setting the trits of Q to μ.

```
1  def bitwise_mul(tnum P, tnum Q):
2
3     tnum sum = tnum(0, 0)
4     # loop runs bitwidth times
5     for i in range(0, bitwidth):
6        tnum product = multiply_bit(P, Q, i)
7        sum = tnum_add(sum, tnum_lshift(product, i))
8     return sum
9
10 def multiply_bit(tnum P, tnum Q, u8 i):
11
12    # Bit position i of tnum P is a certain 0
13    if (P.v[i] == 0 and P.m[i] == 0):
14       return tnum(0, 0)
15    # Bit position i of tnum P is a certain 1
16    elif (P.v[i] == 1 and P.m[i] == 0):
17       return Q
18    # Bit position i of tnum P is uncertain
19    else:
20       # "kill" all bits of Q that are a certain 1,
21       # i.e. set them to uncertain
22       for j in range(0, bitwidth):
23          if (Q.v[j] == 1 and Q.m[j] == 0) :
24             Q.v[j] = 0
25             Q.m[j] = 1
26       return Q
```

Listing 5: Bitwise multiplication (*i.e.,* `bitwise_mul`) by Regehr *et al.* [42]

Setup. We performed all our experiments on the Cloudlab [53] testbed. We used two 10-core Intel Skylake CPUs running at 2.20 GHz for a total of 20 cores, with 192GB of memory.

A. Evaluation of Precision of `our_mul`

We evaluate the precision of `our_mul` compared to `bitwise_mul` and `kern_mul` by exhaustively evaluating all pairs of tnum inputs at a given bitwidth n. We set $n = 8$ to keep the running times tractable.

Consider two abstract tnum multiplication operations $\texttt{op}_1$ and $\texttt{op}_2$. Given two tnums P and Q, suppose $R_1 \triangleq \texttt{op}_1(P, Q)$ and $R_2 \triangleq \texttt{op}_2(P, Q)$. For fixed P and Q, $\texttt{op}_1$ is more precise than $\texttt{op}_2$ if $R_1 \neq R_2 \wedge R_1 \sqsubseteq_{\mathbb{A}} R_2$, or equivalently, $R_1 \neq R_2 \wedge \gamma(R_1) \sqsubseteq_{\mathbb{C}} \gamma(R_2)$, where $\sqsubseteq_{\mathbb{C}}$ is the subset relation $\subseteq$. In general, two such output tnums R_1 and R_2 need not be comparable using the abstract order $\sqsubseteq_{\mathbb{A}}$. For example, at bitwidth $n = 9$, with input tnums $P = 000000011$, $Q = 011\mu011\mu\mu$, the `kern_mul` algorithm produces $R_1 = \mu\mu\mu\mu0\mu\mu\mu\mu$ whereas `our_mul` produces $R_2 = 0\mu\mu\mu\mu\mu\mu\mu\mu$. However, empirically, for tnums of width $n = 8$, outputs R_1 and R_2 turn out to be always comparable using $\sqsubseteq_{\mathbb{A}}$. That is, at $n = 8$, $R_1 \neq R_2 \Rightarrow R_1 \sqsubseteq_{\mathbb{A}} R_2 \vee R_2 \sqsubseteq_{\mathbb{A}} R_1$. Hence, we can simply compare the cardinalities of $\gamma(R_1)$ and $\gamma(R_2)$ as a measure of the relative precision of $\texttt{op}_1$ and $\texttt{op}_2$, for given input tnum pair (P, Q).

Figure 4 compares the cardinalities of $\gamma(R_1)$ and $\gamma(R_2)$ when enumerating every possible input tnum pair (P, Q) of width 8, and only considering cases where $R_1 \neq R_2$. Note that $R_1 \neq R_2$ only when R_1 (similarly R_2) is a tnum with a larger number of unknown trits (μ) than R_2 (similarly R_1). We use a log_2 scale for the x-axis: each tick on the x-axis to the right of 0 is a point where `our_mul` produces a tnum that is more precise in exactly one trit position when compared to the multiplication algorithm it is pitted against. We observe that for around 80% of the cases, `our_mul` produces a more precise tnum than both `kern_mul` and `bitwise_mul` (the data to the right of 0 in Figure 4). In summary, our multiplication is more precise than `kern_mul` and `bitwise_mul`.

Note that `our_mul` and `kern_mul` produce the same result $(R_1 = R_2)$ for 99.92% of all possible 8-trit input tnum pairs. We evaluate how the relative precision between these algorithms varies with increasing bitwidth by enumerating the trends from bitwidth $n = 5$ to $n = 10$. We observe that (1) the fraction of inputs where $R_1 = R_2$ decreases, and (2) `our_mul` produces more precise results than `kern_mul` for a higher fraction of those inputs where the outputs differ $(R_1 \neq R_2)$ but are comparable $(R_1 \sqsubseteq_{\mathbb{A}} R_2 \vee R_2 \sqsubseteq_{\mathbb{A}} R_1)$. The full results are in the extended technical report [49].

Due to the non-associativity of tnum addition (§III-A), some orders of adding tnums are more precise than others, while increasing the number of tnums added makes the result less precise. Hence, the order and number of tnums added is significant to the precision of each multiplication algorithm. In general, `our_mul` is more precise than both `kern_mul` and `bitwise_mul` due to its value-mask decomposition. Both `kern_mul` and `bitwise_mul` add tnums each of which contains both certain and uncertain trits. Due to the value-mask decomposition, `our_mul` postpones such an addition until the very last step of the algorithm. Further, `our_mul` is more precise than `kern_mul` with increasing bitwidth (n), since `our_mul` has fewer additions $(n + 1)$ than `kern_mul` $(2n)$.

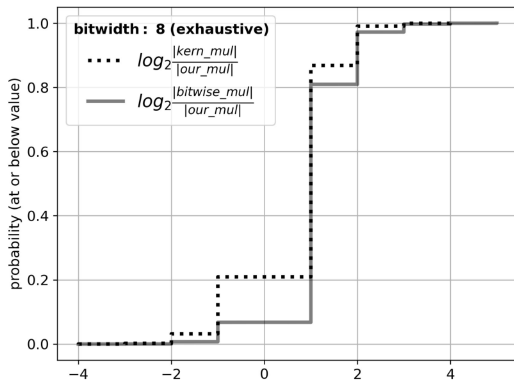

Fig. 4: Cumulative distribution of the ratio of set sizes of the tnums (after concretization) produced by (a) kern_mul to our_mul, and (b) bitwise_mul to our_mul, for cases where the output from our_mul is different. Results presented in log scale (log_2). The input tnums pairs are drawn from the set of all possible tnum pairs of bitwidth 8.

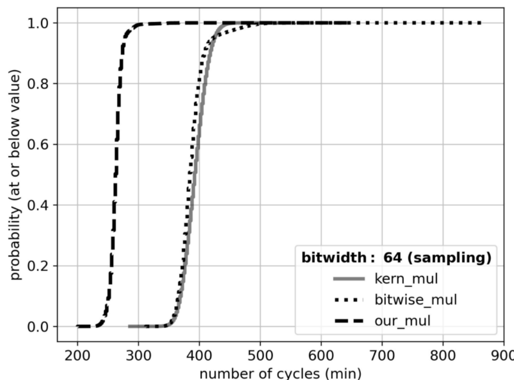

Fig. 5: Cumulative distribution of the minimum number of CPU cycles taken by bitwise_mul, kern_mul, and our_mul for 40 million randomly sampled 64-bit input tnum pairs.

B. Performance Evaluation of our_mul

We compare the performance (in CPU cycles measured using the RDTSC time stamp counter) of all the tnum multiplication algorithms discussed in this paper: kern_mul [39], bitwise_mul [42], and our new algorithm our_mul. We perform the experiment using 40 million randomly sampled tnum pairs (64-bit), repeating each input pair 10 times and measuring the minimum number of cycles across these trials. Figure 5 reports the cumulative distribution of this cycle count across all the sampled inputs. All multiplication algorithms have a loop, and for some algorithms, the number of iterations of the loop depend on number of unknown bits in the input operands. Hence, the number of cycles varies across inputs.

We observe that our_mul is faster (*i.e.,* fewer CPU cycles taken) than all the other versions of tnum multiplication. On average, kern_mul takes around 393 cycles, our optimized version of bitwise_mul takes 387 cycles, and our_mul takes 262 cycles (when we take the average of the minimum of 10 trials for each input tnum pair). Our optimizations to bitwise_mul to use machine arithmetic were important as it improved the performance significantly (*i.e.,* from 4921 cycles to 387 cycles). In summary, efficient use of machine arithmetic and the novel computation and summation of partial products makes our_mul 33% (resp. 32%) faster on average than kern_mul (resp. our optimized version of bitwise_mul).

V. Related Work

BPF safety. Given the widespread use of BPF, recent efforts have explored building safe JIT compilers and interpreters [54, 55, 56, 57]. These works assume the correctness of the in-kernel static checker and the JIT translation happens after the BPF code passes the static checker. Prevail [7] proposes an offline BPF analyzer using abstract interpretation with the zone abstract domain and supports programs with loops. In contrast to this paper, these prior efforts have not looked at verifying the tnum operations in the Linux kernel's static analyzer or explored the tnum domain specifically.

Abstract interpretation. Many static analyses use an abstract domain for abstract interpretation [38, 58, 59]. Abstract domains like intervals, octagons [45], and polyhedra-based domains [60] enhance the precision and efficiency of the underlying operations. Unlike the Linux kernel's tnums, their intended use is in offline settings. Of particular relevance to our work is the known-bits domain from LLVM [44, 3, 43], which, like tnums, is used to reason about the certainty of individual bits in a value. Our work on verifying tnums will be likely useful to LLVM's known-bits analysis, as prior work does not provide proofs of precision and soundness for arithmetic operations such as addition and multiplication.

Safety of static analyzers. One way to check for soundness and precision bugs in static analyzers is to use automated random testing [61, 62]. Recently, Taneja et al. [3] test dataflow analyses in LLVM to find soundness and precision bugs by using an SMT-based test oracle [63]. Bugariu et al. [64] test the soundness and precision of widely-used numerical abstract domains [65, 60]. They use mathematical properties of such domains as test oracles while comparing against a set of representative domain elements. They assume that the oracle specification of operations is correct and precise. This paper differs from these approaches in that we formalize and construct analytical proofs for the abstract operations.

VI. Conclusion

Abstract domains like tnums are widely used to track register values in the Linux kernel and in various compilers. This paper performs verification of tnum arithmetic operations, and develops a new implementation for tnum multiplication. Our algorithm for tnum multiplication is sound, precise, and faster than Linux's kernel multiplication. Our new multiplication algorithm is now part of the Linux kernel.

Acknowledgments

This paper is based upon work supported in part by the National Science Foundation under FMITF-Track I Grant No. 2019302 and the Facebook's Networking Systems Research Award. We thank Edward Cree for his feedback on strength reductions and precision improvements for tnum operations.

REFERENCES

[1] D. Krupp, "Industrial experiences with the Clang analysis toolset," [Online, Retrieved Aug 31, 2021.] https://llvm.org/devmtg/2015-04/slides/Clang_static_analysis_toolset_final.pdf.

[2] J. Corbet, "Gcc and static analysis," [Online, Retrieved Aug 31, 2021.] https://lwn.net/Articles/493599/.

[3] J. Taneja, Z. Liu, and J. Regehr, "Testing static analyses for precision and soundness," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, 2020, pp. 81–93.

[4] GCC Online Documentation, "GCC: Options that control static analysis," [Online, Retrieved Aug 31, 2021.] https://gcc.gnu.org/onlinedocs/gcc/Static-Analyzer-Options.html.

[5] F. Brown, J. Renner, A. Nötzli, S. Lerner, H. Shacham, and D. Stefan, "Towards a verified range analysis for javascript jits," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 135–150.

[6] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan, "Rocksalt: Better, faster, stronger sfi for the x86," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 395–404. [Online]. Available: https://doi.org/10.1145/2254064.2254111

[7] E. Gershuni, N. Amit, A. Gurfinkel, N. Narodytska, J. A. Navas, N. Rinetzky, L. Ryzhyk, and M. Sagiv, "Simple and precise static analysis of untrusted linux kernel extensions," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 1069–1084.

[8] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[9] D. Menendez and S. Nagarakatte, "Alive-infer: Data-driven precondition inference for peephole optimizations in llvm," *SIGPLAN Not.*, vol. 52, no. 6, p. 49–63, Jun. 2017. [Online]. Available: https://doi.org/10.1145/3140587.3062372

[10] J. Schulist, D. Borkmann, and A. Starovoitov, "A thorough introduction to eBPF," [Online, Retrieved Apr 16, 2021.] https://lwn.net/Articles/740157/.

[11] M. Fleming, "Linux Socket Filtering aka Berkeley Packet Filter (BPF)," [Online, Retrieved Apr 16, 2021.] https://www.kernel.org/doc/Documentation/networking/filter.txt.

[12] "Netconf 2018 day 1," [Online, Retrieved Apr 16, 2021.] https://lwn.net/Articles/757201/.

[13] P. Labs, "Instantly troubleshoot your applications on Kubernetes," [Online, Retrieved Apr 16, 2021.] https://pixielabs.ai/.

[14] A. Fabre, "L4drop: Xdp ddos mitigations," [Online, Retrieved Apr 19, 2021.] https://blog.cloudflare.com/l4drop-xdp-ebpf-based-ddos-mitigations/.

[15] N. V. Shirokov, "XDP: 1.5 years in production. Evolution and lessons learned." in *Linux Plumbers Conference*, 2018.

[16] D. Borkmann, "Kube-proxy replacement at the XDP layer," [Online, Retrieved Apr 16, 2021.] https://cilium.io/blog/2020/06/22/cilium-18#kubeproxy-removal.

[17] "Suricata: ebpf and xdp," [Online, Retrieved Apr 16, 2021.] https://suricata.readthedocs.io/en/latest/capture-hardware/ebpf-xdp.html.

[18] "Cilium API-aware networking and security," [Online, Retrieved Apr 14, 2021.] https://cilium.io/.

[19] J. Horn, "Arbitrary read+write via incorrect range tracking in ebpf," [Online, Retrieved Apr 16, 2021.] https://bugs.chromium.org/p/project-zero/issues/detail?id=1454.

[20] V. Palmiotti, "Kernel pwning with eBPF: a love story," [Online, Retrieved August 31, 2021.] https://www.graplsecurity.com/post/kernel-pwning-with-ebpf-a-love-story.

[21] Manfred Paul, "CVE-2020-8835: Linux kernel privilege escalation via improper eBPF program verification," [Online, Retrieved Apr 16, 2021.] https://www.zerodayinitiative.com/blog/2020/4/8/cve-2020-8835-linux-kernel-privilege-escalation-via-improper-ebpf-program-verification.

[22] Rick Larabee, "eBPF and Analysis of the get-rekt-linux-hardened.c Exploit for CVE-2017-16995," [Online, Retrieved Apr 14, 2021.] https://ricklarabee.blogspot.com/2018/07/ebpf-and-analysis-of-get-rekt-linux.html.

[23] L. Nelson, "bpf, x32: Fix bug with alu64 LSH, RSH, ARSH bpf_x shift by 0," [Online, Retrieved Apr 19, 2021.] https://github.com/torvalds/linux/commit/68a8357ec15bdce55266e9fba8b8b3b8143fa7d2.

[24] J. Horn, "bpf/verifier: fix bounds calculation on bpf_rsh," [Online, Retrieved Apr 19, 2021.] https://github.com/torvalds/linux/commit/4374f256ce8182019353c0c639bb8d0695b4c941.

[25] ——, "bpf: fix 32-bit alu op verification," [Online, Retrieved Apr 19, 2021.] https://github.com/torvalds/linux/commit/468f6eafa6c44cb2c5d8aad35e12f06c240a812a.

[26] D. Borkmann, "bpf: Fix incorrect verifier simulation of arsh under alu32," [Online, Retrieved Apr 19, 2021.] https://git.kernel.org/pub/scm/linux/kernel/git/netdev/net.git/commit/kernel/bpf?id=0af2ffc93a4b50948f9dad2786b7f1bd253bf0b9.

[27] ——, "bpf: Fix precision tracking for unbounded scalars," [Online, Retrieved Apr 19, 2021.] https://git.kernel.org/pub/scm/linux/kernel/git/netdev/net.git/commit/kernel/bpf?id=f54c7898ed1c3c9331376c0337a5049c38f66497.

[28] "bpf, x32: Fix bug with ALU64 LSH, RSH, ARSH BPF_X shift by 0," [Online, Retrieved Apr 14, 2021.] https://github.com/torvalds/linux/commit/68a8357ec15bdce55266e9fba8b8b3b8143fa7d2.

[29] J. Horn, "bpf: fix integer overflows," [Online, Retrieved Apr 19, 2021.] https://github.com/torvalds/linux/commit/bb7f0f989ca7de1153bd128a40a71709e339fa03.

[30] "bpf: fix missing error return in check_stack_boundary()," [Online, Retrieved Apr 19, 2021.] https://github.com/torvalds/linux/commit/ea25f914dc164c8d56b36147ecc86bc65f83c469.

[31] "bpf: force strict alignment checks for stack pointers," [Online, Retrieved Apr 19, 2021.] https://github.com/torvalds/linux/commit/a5ec6ae161d72f01411169a938fa5f8baea16e8f.

[32] "bpf: don't prune branches when a scalar is replaced with a pointer," [Online, Retrieved Apr 14, 2021.] https://github.com/torvalds/linux/commit/179d1c5602997fef5a940c6ddcf31212cbfebd14.

[33] D. Borkmann, "bpf: Fix passing modified ctx to ld/abs/ind instruction," [Online, Retrieved Apr 19, 2021.] https://git.kernel.org/pub/scm/linux/kernel/git/netdev/net.git/commit/kernel/bpf?id=6d4f151acf9a4f6fab09b615f246c717ddedcf0c.

[34] J. Stringer, "Document navigating BPF verifier complexity," [Online, Retrieved Aug 31, 2021.] https://github.com/cilium/cilium/issues/5130.

[35] D. Borkmann, "bpf, doc: Add initial nodes on verifier complexity," [Online, Retrieved Aug 31, 2021.] https://github.com/cilium/cilium/commit/ff7c6767180a9923fb1c0646945f29709da6fb6e.

[36] "BPF design Q & A," [Online. Retrieved Jan 20, 2021.] https://www.kernel.org/doc/html/v5.6/bpf/bpf_design_QA.html, 2021.

[37] "BPF: Increase complexity limit and maximum program size," [Online. Retrieved Jul 12, 2021.] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c04c0d2b968ac45d6ef020316808ef6c82325a82, 2019.

[38] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1977, pp. 238–252.

[39] E. Cree, Y. Song, J. Fastabend, and D. Borkmann, "Linux tristate numbers," [Online, Retrieved Aug 31, 2021.] https://github.com/torvalds/linux/blob/v5.10/kernel/bpf/tnum.c.

[40] D. Monniaux, "Verification of device drivers and intelligent controllers: a case study," in *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, 2007, pp. 30–36.

[41] A. Miné, "Abstract domains for bit-level machine integer and floating-point operations," in *WING'12-4th International Workshop on invariant Generation*, 2012, p. 16.

[42] J. Regehr and U. Duongsaa, "Deriving abstract transfer functions for analyzing embedded software," *ACM SIGPLAN Notices*, vol. 41, no. 7, pp. 34–43, 2006.

[43] M. Mukherjee, P. Kant, Z. Liu, and J. Regehr, "Dataflow-based pruning for speeding up superoptimization," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–24, 2020.

[44] "LLVM Known Bits analysis," [Online, Retrieved Apr 20, 2021.] https://www.llvm.org/docs/GlobalISel/KnownBits.html.

[45] A. Miné, "Tutorial on static inference of numeric invariants by abstract interpretation," *Foundations and Trends in Programming Languages*, vol. 4, no. 3-4, pp. 120–372, 2017.

[46] *Sound, Precise, and Fast Abstract Interpretation with Tristate Numbers*. Zenodo, 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5703630

[47] P. Cousot and R. Cousot, "Static determination of dynamic properties of programs," 01 1976.

[48] U. P. Khedker and D. M. Dhamdhere, "A generalized theory of bit vector data flow analysis," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 5, pp. 1472–1511, 1994.

[49] H. Vishwanathan, M. Shachnai, S. Narayana, and S. Nagarakatte, "Sound, precise, and fast abstract interpretation with tristate numbers," *arXiv preprint arXiv:2105.05398*, 2021.

[50] T. Reps, M. Sagiv, and G. Yorsh, "Symbolic implementation of the best transformer," in *In Proc. VMCAI*, 2004, pp. 252–266.

[51] A. Thakur and T. Reps, "A method for symbolic computation of abstract operations," in *Computer Aided Verification*, P. Madhusudan and S. A. Seshia, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 174–192.

[52] A. Thakur, A. Lal, J. Lim, and T. Reps, "Posthat and all that: Automating abstract interpretation," in *The Fourth Workshop on Tools for Automatic Program Analysis (TAPAS)*. Elsevier, January 2013. [Online]. Available: https://www.microsoft.com/en-us/research/publication/posthat-and-all-that-automating-abstract-interpretation/

[53] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. hh0Cecchet, S. Kar, and P. Mishra, "The design and operation of CloudLab," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, Jul. 2019, pp. 1–14. [Online]. Available: https://www.flux.utah.edu/paper/duplyakin-atc19

[54] L. Nelson, J. Van Geffen, E. Torlak, and X. Wang, "Specification and verification in the field: Applying formal methods to bpf just-in-time compilers in the linux kernel," in *Usenix Operating Systems Design and Implementation (OSDI)*, 2020.

[55] J. Van Geffen, L. Nelson, I. Dillig, X. Wang, and E. Torlak, "Synthesizing jit compilers for in-kernel dsls," in *International Conference on Computer Aided Verification*. Springer, 2020, pp. 564–586.

[56] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. Wang, "Scaling symbolic evaluation for automated verification of systems code with serval," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 225–242.

[57] X. Wang, D. Lazar, N. Zeldovich, A. Chlipala, and Z. Tatlock, "Jitk: A trustworthy in-kernel interpreter infrastructure," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 33–47.

[58] P. Cousot, "Lecture 13 notes: Mit 16.399, abstract interpretation," [Online, Retrieved Apr 16, 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.

[59] ——, "Abstract interpretation based formal methods and future challenges," in *Informatics*. Springer, 2001, pp. 138–156.

[60] G. Singh, M. Püschel, and M. Vechev, "Fast polyhedra abstract domain." Association for Computing Machinery, 2017, p. 46–59.

[61] C. Klinger, M. Christakis, and V. Wüstholz, "Differentially testing soundness and precision of program analyzers." Association for Computing Machinery, 2019, pp. 239–250.

[62] P. Cuoq, B. Monate, A. Pacalet, V. Prevosto, J. Regehr, B. Yakobowski, and X. Yang, "Testing static analyzers with randomly generated programs," in *NASA Formal Methods*, A. E. Goodloe and S. Person, Eds. Springer Berlin Heidelberg, 2012, pp. 120–125.

[63] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, G. Lup, J. Taneja, and J. Regehr, "Souper: A synthesizing superoptimizer," *arXiv preprint arXiv:1711.04422*, 2017.

[64] A. Bugariu, V. Wüstholz, M. Christakis, and P. Müller, "Automatically testing implementations of numerical abstract domains," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 768–778.

[65] A. Miné, "The octagon abstract domain," *Higher-Order and Symbolic Computation (HOSC)*, pp. 31–100, 2006.