Program Analysis Combining Generalized Bit-Level and Word-Level Abstractions

GUANGSHENG FAN, National University of Defense Technology, China
LIQIAN CHEN*, National University of Defense Technology, China
BANGHU YIN, National University of Defense Technology, China
WENYU ZHANG, National University of Defense Technology, China
PEISEN YAO, State Key Laboratory of Blockchain and Data Security, China, Zhejiang University, China, and Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, China
JI WANG*, National University of Defense Technology, China

Abstract interpretation is widely used to determine programs' numerical properties. However, current abstract domains primarily focus on mathematical semantics, which do not fully capture the complexities of real-world programs relying on machine integer semantics and involving extensive bit-vector operations. This paper presents a solution that combines a bit-level abstraction and a word-level abstraction to capture machine integer semantics. First, we generalize the bit-level abstraction used in the Linux eBPF verifier for determining known and unknown bits of real-world programs, by supplementing all required operations as a standard abstract domain. Based on this abstraction, we design an abstract domain that is signedness-aware and simultaneously retains both the above bit-level and the word-level bound information. These two levels of information cooperate via a standard reduced product operation to improve analysis precision. We implement the proposed domains in the Crab analyzer and the out-of-kernel eBPF verifier PREVAL. Experiments demonstrate their effectiveness in analyzing SV-COMP benchmark programs, assisting hardware designs, and eBPF verification.

CCS Concepts: • **Software and its engineering → Automated static analysis**.

Additional Key Words and Phrases: Abstract Interpretation, Range Analysis, Abstract Domain

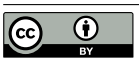
ACM Reference Format:
Guangsheng Fan, Liqian Chen, Banghu Yin, Wenyu Zhang, Peisen Yao, and Ji Wang. 2025. Program Analysis Combining Generalized Bit-Level and Word-Level Abstractions. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA030 (July 2025), 23 pages. https://doi.org/10.1145/3728905

1 INTRODUCTION

Abstract interpretation [7] based static analysis is widely used to determine the numerical properties of programs and ensure program safety. The core element of this framework is the notion of abstract

*Liqian Chen and Ji Wang are the corresponding authors.

Authors' Contact Information: Guangsheng Fan, State Key Laboratory of Complex & Critical Software Environment, College of Computer Science and Technology, National University of Defense Technology, China, guangshengfan@nudt.edu.cn; Liqian Chen, State Key Laboratory of Complex & Critical Software Environment, College of Computer Science and Technology, National University of Defense Technology, China, lqchen@nudt.edu.cn; Banghu Yin, College of Systems Engineering, National University of Defense Technology, China, bhyin@nudt.edu.cn; Wenyu Zhang, State Key Laboratory of Complex & Critical Software Environment, College of Computer Science and Technology, National University of Defense Technology, China, wenyuzhang08@nudt.edu.cn; Peisen Yao, State Key Laboratory of Blockchain and Data Security, China and Zhejiang University, China and Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, China, pyaoaa@zju.edu.cn; Ji Wang, State Key Laboratory of Complex & Critical Software Environment, College of Computer Science and Technology, National University of Defense Technology, China, wj@nudt.edu.cn.



ACM 2994-970X/2025/7-ARTISSTA030
https://doi.org/10.1145/3728905

domain, which can directly impact precision and efficiency. Over the years, various abstract domains have been proposed, such as intervals [8], zones [39], octagons [40], polyhedra [9], etc. Based on mathematical semantics, almost all these conventional domains utilize unbounded integers with infinite precision to model integer program variables. Meanwhile, they usually only design abstractions for arithmetic operations but have limited support for bit-vector operations [31].

However, real-world programs use integer types defined as bounded machine integers, i.e., bit-vectors, with finite precision. Machine integers can be represented as a circle, as in Fig. 1. By drawing a straight line from the N pole and S pole, the circle is divided into two hemispheres: the 0-hemisphere, which ranges from the minimum unsigned integer 00...00(*umin*) to the maximum signed integer 01...11(*smax*) clockwise, and the 1-hemisphere, which ranges from the minimum signed integer 10...00(*smin*) to the maximum unsigned integer 11...11(*umax*) clockwise. Fig. 1 also provides an example of 8-bit integers, where unsigned integer representations are shown in green and signed integer representations in orange.

A comprehensive static analysis based on machine integer semantics must address two challenges:

- *Characterizing wrapped behaviors*: Word-level arithmetic operations are inherently modular under machine integer semantics. Hence, adding two positive signed integers can overflow, yielding a negative value. Unfortunately, many existing abstract domains in program analysis are limited to producing only positive results, thereby failing to capture such wrapped behaviors. This oversight can lead to missed opportunities for detecting critical bugs. For example, numerous CWEs are related to machine integer behaviors [10–18].
- *Modelling bit-vector operations*: Bit-vector operations are prevalent in real-world programs, particularly in cryptographic applications and low-level embedded systems. Additionally, when software analyzers are employed for hardware verification, the process often involves translating Verilog RTL into corresponding C or LLVM programs [1, 43, 44, 58], which frequently contain numerous bit-vector operations. The accuracy with which these operations are modelled directly impacts the overall precision of the program analysis.

To address these challenges, there are two categories of techniques for designing abstract domains.

- One approach is to extend existing word-level abstractions to incorporate machine integer semantics. This requires designing abstractions for bit-vector operations from an inherently complex word-level perspective, such as the wrapped interval domain (`wint`) [21, 47]. However, the complexity of these algorithms led to undetected soundness bugs that persisted for nearly a decade [32]. Furthermore, relying solely on interval abstraction can result in a loss of precision, as this word-level approach captures only the properties of the entire bit vector, neglecting the finer-grained properties of individual bits.
- Another approach is to augment conventional domains with modular arithmetic and apply bit-level abstractions specifically for bit-vector operations. For instance, Astrée [4] extends the interval domain with a modular component and employs a bitfield domain to analyze bit-vector operations [41]. While this bit-level abstraction can be implemented efficiently, it still cannot track precise bit-level information for general arithmetic operations, limiting its precision in specific contexts.

The eBPF verifier the Linux kernel takes a step forward, combining both word-level and bit-level abstractions to ensure eBPF program safety [60, 61]. The former includes four interval-like abstractions, i.e., (un)signed 64/32-bit intervals, while the latter is bitwise and named `tnum`, which determines known and unknown bits. The `tnum` abstraction is applied to bit-vector operations and several arithmetic operations. However, there are still other word-level operations that it cannot deal with, such as the division operation and the greater-than-condition judgment. Furthermore, since the eBPF verifier is designed to track each path of the restricted loop-free program independently,

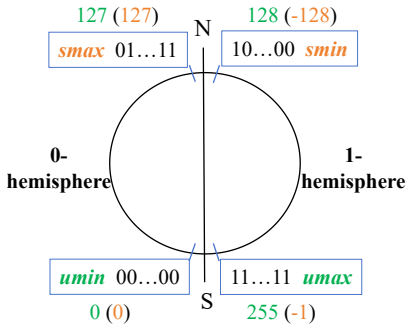


Fig. 1. Machine integer representation

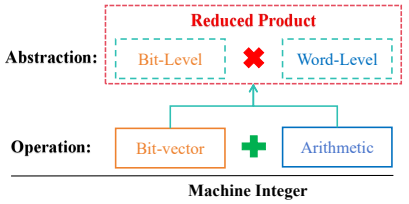


Fig. 2. Analysis Framework

the eBPF abstraction has neither join, meet, nor widening operations designed, which are important for analyzing general real-world programs.

Inspired by the program analysis in the Linux eBPF verifier, this paper presents an efficient and practical program analysis framework that combines word-level and bit-level abstractions for both arithmetic and bit-vector operations, as shown in Fig. 2, in generic real-world programs.

- First, we generalize the `tnum` abstraction in the eBPF verifier by complementing all essential domain operations, resulting in an independent domain $\texttt{tnum}^+$ to evaluate generic real-world programs, e.g., without the loop-free limitation. Our bit-level abstraction advances by retaining known and unknown bits in each signedness case of a variable, considering the features of machine integer distribution. As seen in Fig. 1, there is a significant difference in integers located on both sides of the N (S) Pole and adjacent to it. Abstraction for these integers using $\texttt{tnum}^+$ may lead to many unknown bits, e.g., we cannot determine any bits for a valuable $v \in \{umin, umax\}$.
- Then, we propose a word-level abstraction that records the lower and upper bound of the potential values also in each hemisphere, making it more precise than the existing `wint` domain [21, 47], which combines the values in both signs. Meanwhile, our word-level abstraction can play an equal role as the combination of the four interval-like abstractions in the eBPF verifier.
- Finally, we design a domain named `swb`, which combines the above bit-level and word-level abstractions. The combination is via a reduced product operation based on the key insight that the upper (lower) abstraction at the word level for each sign is obtained by setting all unknown bits to 1 (0). We further establish that this reduced product operation is standard, as it is both sound and capable of computing the most precise abstraction at both the word and bit levels for any program operation.

We have implemented the above abstract domains in the Crab library [28] and the Linux eBPF verifier PREVAIL [23] which has recently developed to be the eBPF verifier for Windows [19]. Experiments have shown the promising ability of our domains to analyze real-world C programs (with loops), bit-vector dense programs translated from hardware verification [1], and eBPF programs. In summary, we make the following main contributions:

- We generalize the `tnum` abstraction in the eBPF verifier to support real-world program analysis.
- We propose an abstract domain named `swb` that retains both (1) known and unknown bits at the bit-level and (2) interval bounds at the word-level in different hemispheres. They cooperate by a proven standard of reduced product operation.

- We implement our abstract domains in the Crab library and eBPF verifier PREVAIL. Experiments have shown their promising ability to do analysis of real-world programs, hardware verification, and eBPF verification.

2 PRELIMINARY

Abstract interpretation [7] is a general theory of the approximation of formal program semantics. It replaces the complex computation in the concrete domain $\mathcal{D}$ by an approximate computation in the abstract domain $\mathcal{D}^\sharp$. The soundness and precision are reasoned by the Galois connection.

*Definition 2.1 (**Galois connection**).* Given two posets $(\mathcal{D}, \subseteq)$ and $(\mathcal{D}^\sharp, \sqsubseteq)$, the pair $(\alpha\colon \mathcal{D} \to \mathcal{D}^\sharp, \gamma\colon \mathcal{D}^\sharp \to \mathcal{D})$ is a Galois connection if: $\forall c \in \mathcal{D}, a \in \mathcal{D}^\sharp, \alpha(c) \sqsubseteq a \iff c \subseteq \gamma(a)$, which is denoted as $(\mathcal{D}, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{D}^\sharp, \sqsubseteq)$, where α and γ correspondingly are called the abstraction and concretization functions [42].

Combining Multiple Abstract Domains [6]. Consider two domains: $\mathcal{D}_1^\sharp$ and $\mathcal{D}_2^\sharp$, with abstraction functions α_1 and α_2 and concretization functions γ_1 and γ_2, respectively. The Cartesian product domain $\mathcal{D}_{1\times2}^\sharp \triangleq \mathcal{D}_1^\sharp \times \mathcal{D}_2^\sharp$ combines the expressive power of its component domains, but each component domain conducts analysis independently, resulting in little improvement in analysis precision. To address this issue, the concept of reduced product is proposed. For an abstract value (a_1, a_2) in $\mathcal{D}_{1\times2}^\sharp$, the reduce operator $\rho : \mathcal{D}_{1\times2}^\sharp \to \mathcal{D}_{1\times2}^\sharp$ improves the precision of an abstract value in one domain by incorporating information from another. It aims to find the optimal abstract value for each component domain by reducing it to the smallest possible value that aligns with the concretization of the paired abstract value, i.e., $\rho(a_1, a_2) \triangleq (\alpha_1(\gamma_{1\times2}(a_1, a_2)), \alpha_2(\gamma_{1\times2}(a_1, a_2)))$.

Bit-Level Abstraction in Linux eBPF Verifier. The `tnum` abstraction [60] in the eBPF verifier tracks each bit of variables drawn from the set of n-bit machine integers $\mathcal{B}_n$, i.e., the set of all bit-vector of width n. Each bit can be known (0 or 1) or unknown (μ). We use a pair of two n-width bit-vector (v, m) denoted as $(value, mask)$ in the eBPF verifier to represent an abstract element in the `tnum` domain $\mathcal{D}_t^\sharp$. In this way, for $P = (v, m) \in \mathcal{D}_t^\sharp$, we can get:

$$\forall i \in [0, n-1].\ P[i] = \begin{cases} 0 & \text{if } P.v[i] = 0 \wedge P.m[i] = 0 \\ 1 & \text{if } P.v[i] = 1 \wedge P.m[i] = 0 \\ \mu & \text{if } P.v[i] = 0 \wedge P.m[i] = 1 \end{cases}$$

By this end, the known bits are recorded by v, while the unknown bits are by m. For example, for $x \in \{8, 9, 10, 11\}$, the `tnum` abstraction is $0b10\mu\mu$, with $v = 0b1000$ and $m = 0b0011$. This representation also captures some word-level information. The minimum value of the `tnum` abstraction is v, setting all unknown bits to 0. Also, the maximum abstraction is $v + m$, setting all unknown bits to 1. Note that `tnum` is indeed fixed to 64-bit representation in the kernel. Also, this representation differs slightly from the KnownBits abstraction used in LLVM, which encodes known bits utilizing a pair of n-bit values $(zero, one)$ to track bits that are certainly 0 and 1, respectively [59].

The abstract value is the bottom, denoted by $\bot_t$, when there exists any undefined bit violating the above representation rule, i.e., $v \,\&\, m \neq 0$; and is top, denoted by $\top_t$, when all bits are unknown, i.e., $v = 0$ and $m = umax$. For any two abstract elements that are not bottom, the inclusion testing can be defined as: $P \sqsubseteq_t Q \triangleq (P.v \,\&\, {\sim}Q.m) = Q.v$.

The Galois connection between the concrete domain of powerset of n-width bit-vectors $\mathcal{B}_n$ and the `tnum` abstract domain can be defined as: $(\wp(\mathcal{B}_n), \subseteq) \underset{\alpha_t}{\overset{\gamma_t}{\leftrightarrows}} (\mathcal{D}_t^\sharp, \sqsubseteq_t)$, where $\mathcal{D}_t^\sharp$ is the set of all tnums over $\mathcal{B}_n$: $\{(v, m) \mid v \in \mathcal{B}_n, m \in \mathcal{B}_n\}$. The abstraction function $\alpha_t : \wp(\mathcal{B}_n) \to \mathcal{B}_n \times \mathcal{B}_n$ is

defined as:

$$\alpha_{\&}(B) \triangleq \&\{b \mid b \in B\} \quad \alpha_{|}(B) \triangleq |\{b \mid b \in B\}$$
$$\alpha_t(B) \triangleq \left(\alpha_{\&}(B), \alpha_{\&}(B) \oplus \alpha_{|}(B)\right)$$

where $B \subseteq \mathcal{B}_n$ is a subset of n-width bit-vectors.

The concretization function $\gamma_t : \mathcal{B}_n \times \mathcal{B}_n \to \wp(\mathcal{B}_n)$ is:

$$\gamma_t(\bot_t) \triangleq \varnothing \quad \gamma_t(\top_t) \triangleq \mathcal{B}_n$$
$$\gamma_t(P) \triangleq \{c \in \mathcal{B}_n \mid c \,\&\, {\sim}P.m = P.v\}$$

In the kernel, `tnum` implements abstractions for all bit-vector operations and arithmetic operations except div, mod, and neg. Meanwhile, all abstractions of shift operation only consider shift constant numbers, e.g., lsh 2 bits. We denote these existing operations over $\mathcal{D}_t^\sharp$ as: $+_t$, $-_t$, $\times_t$, $|_t$, $\&_t$, $\oplus_t$, $\ll_t$, $\gg_{lt}$ and $\gg_{at}$. We refer to [60] for more details.

Abstraction Combination in Linux eBPF Verifier. Despite the `tnum` abstraction, there are another four word-level abstractions in the eBPF verifier, i.e., (un)signed 64/32-bit interval abstractions. Rather than using the standard reduced products, almost all of these five abstractions are combined via the so-called *Reduction/abstract* operators, which computes an abstraction based on the analysis results from other abstractions [61]. The kernel uses the `reg_bounds_sync` function as a shared "tail" of computation in all of its *Reduction/abstract* operators. This function has four steps: (1) improves the signed and unsigned interval abstractions based on the `tnum` abstraction; (2) updates unsigned and signed interval abstractions by exchanging information between them; (3) improves the `tnum` based on the unsigned interval abstractions; (4) repeats step (1).

We just describe much more about step (3) here. for an unsigned 64-bit interval abstraction $[min, max]$, its corresponding `tnum` abstraction can be derived by the *tnum_range* function. It first determines the highest bit l where min differs from max. Then, all bits starting from l and lower than l will be set to unknown in the `tnum` abstraction, while the other higher bits are known as min and max. For example, the `tnum` abstraction of the interval ranging from $0b1001$ to $0b1111$ is $(0b1000, 0b0111)$, i.e., $0b1uuu$.

3 MOTIVATION

We illustrate our motivation with a program taken from [47] whose input is replaced with $y = -8$ and $x = 5$, i.e., $0b00000101$, as shown in Fig. 3. Due to the machine integer characteristics, the value of x will wrap to be negative after multiple times of adding 8, i.e., $0b00001000$, and should undoubtedly be -123 in line 7.

Limitations of Existing Abstract Domains. Based on mathematical integer semantics, the traditional interval domain has infinite precision. After several iterations of the loop, it determines that x is always greater than or equal to 5 and can be positive infinity. Consequently, the loop condition will never be violated, meaning that line 7 cannot be reached. This outcome is incorrect.

In contrast, the wrapped interval domain (`wint`) [21, 47] is based on the machine integer semantics and can identify that x may wrap to be negative after the loop execution. However, only characterizing word-level information is not enough to verify this program. As shown in Table 1, when using the `wint` domain, we can only derive that x is in the range $(\!|-128, -121|\!)$ leading to false positives in the assertion judgment. This is because `wint` only determines that x may be any integer in interval $(\!|5, 127|\!)$ before line 5, which after adding 8 would be $(\!|13, -121|\!)$. Then, joining this interval with the previous one, `wint` can derive $x \in (\!|5, -121|\!)$ at line 4, which can reach convergence after several times iterations of analysis. Finally, once the loop condition is unsatisfied, x is in the range of $(\!|-128, -121|\!)$.

```
1 char x, y;
2 y = -8;
3 x = 5;//x=(x|5)&(-3)
      ;
4 while(x >= y){
5     x = x - y;
6 }
7 assert(x == -123);
```

Fig. 3. Motivating program

Table 1. Program invariants of x derived by `wint` and `swb` domain at each location (right) for Fig. 3

L	wint	swb
4	$(\!\|5, -121\|\!)$	$\langle(\!\|5, 125\|\!), (\!\|-123, -123\|\!)\rangle \wedge \langle(5, 120), (-123, 0)\rangle$
5	$(\!\|5, 127\|\!)$	$\langle(\!\|5, 125\|\!), \bot_i\rangle \wedge \langle(5, 120), \bot_t\rangle$
6	$(\!\|13, -121\|\!)$	$\langle(\!\|13, 125\|\!), (\!\|-123, -123\|\!)\rangle \wedge \langle(5, 120), (-123, 0)\rangle$
7	$(\!\|-128, -121\|\!)$	$\langle\bot_i, (\!\|-123, -123\|\!)\rangle \wedge \langle\bot_t, (-123, 0)\rangle$

Our Abstract Domain swb. The `wint` domain focuses solely on word-level abstraction. In contrast, this paper introduces a new abstract domain, `swb`, which integrates both word-level and bit-level abstractions. This combined approach enables the successful verification of the program. Table 1 compares the program invariants derived from `swb` with those from `wint`. In this table, the invariants preceding the $\wedge$ symbol correspond to word-level abstraction, while those following it represent bit-level abstraction.

Benefit of Bit-level Abstraction in swb. The bit-level abstraction in `swb` domain generalizes the `tnum` abstraction in the eBPF verifier to support analyzing programs with loops. It can retain known and unknown bits in each hemisphere. This is very useful in improving the precision of the bit-vector program analysis. As we can observe in the program, no matter how many times adding 8 is made, the lowest three bits of x should remain unchanged and always be "101", because the lowest three bits of 8 are "000", which cannot affect those corresponding bits of x during the addition operation, e.g., $0b1000 + 0b0101 = 0b1101$. On the other hand, adding 8 only affects the other bits, e.g., $0b1000 + 0b01111101 = 0b10000101$. With the help of the bit-level abstraction in `swb`, we can determine the possible positive value of x before line 5 should be those integers ranging from -3 ($0b11111101$) to 125 ($0b01111101$) and have a trailing three bits "101", because -3 and 125 are the minimal and maximal integer greater than y, respectively.

Being signedness-aware here is also notable for precision analysis. -3 is the only negative integer that belongs to the above integers and is only different from 125 in the sign bit. However, when using the `tnum` domain, the direct abstraction of the above integers with the same trailing can immediately make all the highest five bits unknown, i.e., $0buuuuu101$. Its corresponding concretization is the set of all possible integers with the trailing "101", which is far less precise than the signedness-aware bit-level abstraction in our `swb` domain.

Benefit of Combining Bit-level and Word-level Abstractions in swb. To maintain consistency with the signedness-aware bit-level abstraction, the word-level abstraction in `swb` is designed to preserve interval range abstraction for each signedness case. In the program shown in Fig. 3, it can only derive a similar abstraction to the `wint` domain for the variable x. For example, before line 5, it can also determine that x is in the range of $(\!|5, 127|\!)$.

Combining the word-level and bit-level abstractions in `swb` via standard reduced product operation can improve the whole analysis precision. In line 5, we can improve the word-level abstraction of x to the interval $(\!|5, 125|\!)$, since we know x is less than 125, with the help of the bit-level abstraction. Meanwhile, we can improve the bit-level abstraction to $(5, 120)$, i.e., positive integers with the trailing "101", because -3 does not satisfy the word-level abstraction. In other words, using the `swb` domain, we can determine that x is those integers ranging from 5 to 125 and with a trailing "101". Thus, after adding 8, x may wrap from 125 to -123, i.e., $0b10000101$, the only integer value that does not satisfy the loop condition, leading to the assertion becoming true in line 7.

Comparing swb with Existing Abstract Domains. When we replace the assignment of x at line 3 in Fig. 3 with a more complex expression consisting of bit-vector operations, i.e., $(x|5)\&(-3)$, x can take on any integer with its lowest three bits set to "101". The assertion should still be true.

Unfortunately, the `wint` domain implemented in the Crab library still reports a false positive here. After line 3, it considers x to be top, i.e., any integers. Thus the abstraction of x before line 5 would be $(\!|-8, 127|\!)$, and after adding 8 would be $(\!|0, -121|\!)$. However, joining this abstraction with the former top state would remain top before line 4. Therefore, when the loop condition is not satisfied, `wint` domain can only determine that the range of x is $(\!|-128, -9|\!)$, which is much more imprecise than the analysis result of the program before modification.

Moreover, the machine integer analysis method implemented in Astrée [4, 41] also cannot prove the assertion. Although it can model the bit-vector operation in line 3, it just transforms the bit-level abstraction result into word-level abstraction after line 3, thus ignoring all bit-level information useful for subsequent analysis. It also just focuses on word-level operations for the loop like `wint`.

Fortunately, thanks to the reduced product between word-level and bit-level abstractions, the abstraction of x by our `swb` domain in line 5 would still be $(\!|5, 125|\!)$ and has the trailing "101", thus validating the assertion.

4 BIT-LEVEL ABSTRACTION

This section presents the bit-level abstraction of `swb`, which retains known and unknown bits in each signedness case. It generalizes the `tnum` abstraction used in the eBPF verifier and extends to be signedness-aware. For the sake of space, we only present some key points of the bit-level abstraction. Their full details are available in the supplementary material.

4.1 Generalizing `tnum` Abstraction

We generalize `tnum` to support all the necessary operations in Table 2 and transfer functions as a standard abstract domain. We denote this generalized `tnum` domain as $\texttt{tnum}^+$.

Lattice Operations. To form the complete lattice of $\mathcal{B}_n$, the meet ($\sqcap_t$) and join ($\sqcup_t$) operations are critical components. However, since the kernel tracks along each execution path for loop-free programs, these operations are not implemented in the eBPF verifier. On the other hand, they are critical for extending the applicability of the `tnum` domain. To design algorithms for the meet and join operations, we should first consider their bit-wise operations:

$$0 \sqcap 0 \triangleq 0,\ 0 \sqcap 1 \triangleq \bot,\ 1 \sqcap 1 \triangleq 1,\ 1 \sqcap \mu \triangleq 1,\ 0 \sqcap \mu \triangleq 0,\ \mu \sqcap \mu \triangleq \mu$$
$$0 \sqcup 0 \triangleq 0,\ 0 \sqcup 1 \triangleq \mu,\ 1 \sqcup 1 \triangleq 1,\ 1 \sqcup \mu \triangleq \mu,\ 0 \sqcup \mu \triangleq \mu,\ \mu \sqcup \mu \triangleq \mu$$

Note that whenever the meet operation on a bit returns $\bot$, the whole result will be $\bot_t$. Based on the bit-wise operations, the meet and join operations of $\texttt{tnum}^+$ are designed as:

$$P \sqcap_t Q \triangleq (P.v \mid Q.v,\ P.m \,\&\, Q.m) \qquad P \sqcup_t Q \triangleq (P.v \,\&\, Q.v,\ P.m \mid Q.m \mid \delta)$$

where $\delta = (P.v \,\&\, \eta) \oplus (Q.v \,\&\, \eta)$, and $\eta = \sim (P.m \mid Q.m)$. η represents those bits that both P and Q are known, thus δ determines the bits that both P and Q are known, but with different values. For $P \sqcap_t Q$, unknown bits of the result must be unknown in P and Q, while all bits with value 1 of P or Q must also be 1 after meeting. In addition, when δ is non-zero, $P \sqcap_t Q$ is bottom, since there are some known bits in conflict values. On the other hand, for $P \sqcup_t Q$, the unknown bits of the result are those that are unknown in P or Q and those that have different values in P and Q, while any bits with a known value of 1 in both P and Q must also be 1 in the result after joining.

Example 4.1. When computing the meet result of $0b1\mu\mu1 = (0b1001, 0b0110)$ and $0b1\mu0\mu = (0b1000, 0b0101)$, we can determine that the unknown bits of the result are $0b0110 \,\&\, 0b0101 = 0b0100$, and the values of the other known bits are $0b1001 \mid 0b1000 = 0b1001$, so the meeting

```
def tnum_udiv(tnum P, tnum Q):
    if Q.v = 0 return ⊤_t
    B_n MaxP := P.v + P.m
    B_n MaxR := MaxP /u Q.v
    tnum R := ⊤_t
    u64 Leadz:=countLeadingZero(MaxR)
    clearHighBits(R.v, Leadz)
    clearHighBits(R.m, Leadz)
    return R
```

Fig. 4. Unsigned division of tnum⁺

Table 2. Supplementary domain operations of tnum⁺ compared to tnum

Type	New-added operations
Lattice	meet, join
Arithmetic	(un)signed division, (un)signed remainder
Bit-vector	truncation, signed extension, zero extension, shift non-constant bits
Extrapolation	widening

result is $0b1\mu01 = (0b1001, 0b0100)$. As for the join result of $0b1\mu01 = (0b1001, 0b0100)$ and $0b0\mu0\mu = (0b0000, 0b0101)$, we can determine that the unknown bits of the result are $0b0100 \mid 0b0101 \mid 0b1000 = 0b1101$, and the values of the other known bits are $0b1001 \,\&\, 0b0000 = 0b0000$, so the joining result is $0b\mu\mu0\mu = (0b0000, 0b1101)$. Additionally, we can determine that the known but conflicting bits of $0b1\mu01$ and $0b0\mu0\mu$ are $0b1000$, i.e, the highest bit is in conflict, thus the meet result of the two abstractions is bottom.

Widening Operation. The lattice height of $\mathcal{D}_t^\sharp$ for n-bits integers $\mathcal{B}_n$ is a finite constant number n, which means the fixpoint iterations of the abstract interpretation-based analysis will eventually get convergence after finite iterations of worst-case n. Therefore, the join operation could replace the widening operation $\triangledown_t$ of $\mathcal{D}_t^\sharp$. To accelerate the fixpoint iteration, we also carefully design $\triangledown_t$ for a common situation. When P and Q have the same trailing known bits but Q has more unknown subsequent higher bits than P, $P \triangledown_t Q$ retains the same trailing known bits but make all other bits unknown.

Example 4.2. Using tnum⁺ domain to analyze the program in Fig. 3, at line 4, we could conduct $0b0000u101 \triangledown_t 0b000uu101$. The above design helps us immediately achieve the widening result of $0buuuu101$, which requires multiple joining attempts to reach.

Arithmetic and Bit-Vector Operations. To generalize the tnum domain, we add abstractions for some essential arithmetic operations, such as unsigned division, signed division, unsigned remainder, signed remainder, and negation. Implementations of these operations are inspired by the KnownBits abstraction [59] in LLVM. Additionally, we have designed abstractions for useful bit-vector operations like truncation, signed extension, and zero extension, which are often used in typecasting. Furthermore, we have included shift operations that can be performed on constant numbers and variables. We briefly describe the unsigned division and truncation here, leaving details of other operations in the supplementary material.

Note that designing a precise enough algorithm for $P /_{ut} Q$ in $\mathcal{D}_t^\sharp$ is challenging. Instead, we determine how many high bits of the quotient R must be 0, which can be easier and more efficient. As shown in Fig. 4, the determination is based on the maximal abstraction of R, which can be derived by $(P.v + P.m) \,/_u\, Q.v$, since $P.v + P.m$ denotes the upper bound of P, while $Q.v$ is the lower bound of Q. All bits in R that are higher than the most significant bit in $max(R)$ must be set to 0, which means the corresponding bits in $R.v$ and $R.m$ must be set to 0 as well. Note that we cannot determine the lower trailing 0 (or 1) bits of R by computing its minimal value because those values greater than the minimal value may have different values in these bits.

Example 4.3. Suppose $P = 0b01\mu0$ and $Q = 0b001\mu$, we first derive that the maximal possible quotient is $max(R) = 0b0110 \,/_u\, 0b0010 = 0b0011$, thus all concrete values of the quotient can be abstracted as $R = 0b00\mu\mu = (0b0000, 0b0011)$. On the other hand, the minimal possible quotient is

$min(R) = 0b0100 /_u 0b0011 = 0b0001$. However, there may be some concrete values of the quotient, e.g., $0b0010$, which can be zero in the lowest bit, though greater than $min(R)$. Therefore, we cannot determine a certain value of the lowest bit. Fortunately, once we get the highest bit in value 1 of $max(R)$, we can soundly determine that the highest two bits of the quotient cannot be 1. Otherwise, there will be some possible values greater than the maximal one, which is a clear paradox.

When we truncate a bit-vector from n bits to $k < n$ bits, we preserve only the lower k bits of the original bit-vector. Therefore, to truncate an abstract element P, we can truncate $P.v$ and $P.m$ respectively using the formula $trunc_t(P, k) = (trunc(P.v, k), trunk(P.m, k))$. This algorithm is simpler and more precise than the one designed in the word-level `wint` domain [21, 47] because `wint` only considers two common cases, while leaving other cases to return top abstraction. This difference can also serve as an example to illustrate how bit-level abstractions outperform word-level abstractions when dealing with bit-vector operations.

Example 4.4. Suppose $P = 0b\mu\mu\mu1 = (0b0001, 0b1110)$, then $trunc_t(P, 2) \triangleq 0b\mu1$, which can be derived by $trunc(0b0001, 2) = 0b01$ and $trunc(0b1110, 2) = 0b10$. For this situation, `wint` only returns top.

Supplemental Transfer Functions. The assignment transfer function can be easily defined with all the above arithmetic and bit-vector abstract operations. We mainly introduce how to deal with the analysis of conditional statements here. Note that comparison statements in LLVM must be signed or unsigned explicitly; thus, we should consider these two cases when computing the abstraction result of the comparison. Take $P \leq_u Q$ as an example, where $P, Q \in \mathcal{D}_t^\sharp$. We compute their corresponding results P' and Q' as:

$$P' \triangleq \begin{cases} \bot_t & \text{if } Q = \bot_t \vee P.v > Q.v + Q.m \\ P & \text{if } Q.v + Q.m = umax \\ P \sqcap_t tnum\_range\,(P.v,\ Q.v + Q.m) & \text{otherwise} \end{cases}$$

$$Q' \triangleq \begin{cases} \bot_t & \text{if } P = \bot_t \vee P.v > Q.v + Q.m \\ Q & \text{if } P.v = umin \\ Q \sqcap_t tnum\_range\,(P.v,\ Q.v + Q.m) & \text{otherwise} \end{cases}$$

where the auxiliary function $tnum\_range(l, u)$ implemented in the Linux kernel, compute the `tnum`$^+$ abstraction from an interval with a lower bound l and upper bound u. The implementation of the signed comparison can be designed similarly. Note that the way we deal with condition statements is different from those in the kernel for `tnum` abstraction, where they only update `tnum` using the interval information by *reg_bound_sync* [61].

Example 4.5. Given $P = 0b\mu10\mu = (0b0100, 0b1001)$ and $Q = 0b01\mu\mu = (0b0100, 0b0011)$, the condition $P \leq_u Q$ can be true since the minimal value of P (i.e, $0b0100$), is less than the maximal value of Q (i.e., $0b0111$). In this case, P will be updated to the overlapping range (i.e., $0b010\mu$) of its previous abstraction and the abstraction of the range from $0b0100$ to $0b0111$ (i.e., $0b0\mu\mu\mu$).

4.2 Signedness-aware Extension

From the machine integer representation in Fig. 1, it is obvious that the `tnum`$^+$ abstraction for those bit-vectors across the South Pole or North Pole, i.e., with different signedness, will lose precision greatly. For example, $0 \sqcup_t -1 = \top_t$. To address the problem, we improve `tnum`$^+$ with signedness, leading to a the bit-level abstraction of `swb`, denoted as $\mathcal{D}_s^\sharp$. The main idea is to maintain two `tnum`$^+$ elements for each variable, to track the `tnum`$^+$ abstraction of the variable in 0-hemisphere and 1-hemisphere respectively.

Domain Representation. We use $\langle T_0, T_1\rangle$ to represent the bit-level abstract value of a variable x, which means that $x \in T_0$ or $x \in T_1$, where T_0 denotes $\bot_t$ or 0-hemisphere tnums and T_1 denote $\bot_t$ or 1-hemisphere tnums. In particular, we use $\bot_s$ to represent the empty bit-level abstract value, sometimes also denoted as $\langle \bot_t, \bot_t\rangle$. Also, we use $\top_s$ to represent $\langle (umin, smax), (smin, umax)\rangle$ which contains all possible values.

The Galois connection between the concrete domain of powerset of n-width bit-vectors $\mathcal{B}_n$ and the bit-level abstraction of swb can be defined as:$(\wp(\mathcal{B}_n), \subseteq) \underset{\alpha_s}{\overset{\gamma_s}{\leftrightarrows}} (\mathcal{D}_s^\sharp, \sqsubseteq_s)$ where $\mathcal{D}_s^\sharp$ is the set of all signed tnums over $\mathcal{B}_n$: $\{\langle (v_0, m_0), (v_1, m_1)\rangle \mid v_0, m_0, v_1, and\ m_1 \in \mathcal{B}_n,\ umin \le v_0, m_0, m_1 \le smax,\ smin \le v_1 \le umax\}$. $\mathcal{D}_s^\sharp$ forms a complete lattice $(\mathcal{D}_s^\sharp, \sqsubseteq_s, \sqcap_s, \sqcup_s, \bot_s, \top_s)$.

The abstraction function $\alpha_s : \wp(\mathcal{B}_n) \to \mathcal{D}_s^\sharp$ is defined as:

$$\alpha_s(B) \triangleq \begin{cases} \bot_s & \text{if } B = \varnothing \\ \langle \alpha_t(B),\ \bot_t\rangle & \text{if } \forall b \in B, umin \le b \le smax \\ \langle \bot_t,\ \alpha_t(B)\rangle & \text{if } \forall b \in B, smin \le b \le umax \\ \langle \alpha_t(B_0),\ \alpha_t(B_1)\rangle & \text{otherwise} \end{cases}$$

where $B \subseteq \mathcal{B}_n$ is a subset of n-width bit-vectors, and $B_0 \triangleq \{b \in B \mid umin \le b \le smax\}$ while $B_1 \triangleq \{b \in B \mid smin \le b \le umax\}$.

The concretization function $\gamma_s : \mathcal{D}_s^\sharp \to \wp(\mathcal{B}_n)$ is defined as:

$$\gamma_s(P) \triangleq \begin{cases} \varnothing & \text{if } P = \bot_s \\ \gamma_t((v, m)) & \text{if } P = \langle \bot_t,\ (v, m)\rangle \vee P = \langle (v, m),\ \bot_t\rangle \\ \gamma_t((v_0, m_0)) \cup \gamma_t((v_1, m_1)) & \text{if } P = \langle (v_0, m_0),\ (v_1, m_1)\rangle \end{cases}$$

where $P \in \mathcal{D}_s^\sharp$ represents an abstract element in the bit-level abstract domain of swb.

Besides, for a $\texttt{tnum}^+$ element T, we can construct its corresponding bit-level abstraction in swb by:

$$split(T) \triangleq \begin{cases} \top_s, & \text{if } T = \top_t \\ \bot_s, & \text{if } T = \bot_t \\ \langle T, \bot_t\rangle & \text{if } T.v \le smax \wedge T.m \le smax \\ \langle \bot_t, T\rangle & \text{if } smin \le T.v \wedge T.m \le smax \\ \langle (T.v,\ T.m \,\&\, smax),\ (T.v \mid smin,\ T.m \,\&\, smax)\rangle & \text{otherwise} \end{cases}$$

Example 4.6. Given concrete values $\{0b1000, 0b1010, 0b0000, 0b0001\}$, their $\texttt{tnum}^+$ abstraction is $0b\mu0\mu\mu$, which also includes 4 redundant concrete values. However, in $\mathcal{D}_s^\sharp$, we can abstract the given values in different hemispheres and derive $\langle 0b000\mu, 0b10\mu0\rangle$, which still keep completeness.

Domain Operators. Assuming $\langle T_0,\ T_1\rangle$ and $\left\langle T_0',\ T_1'\right\rangle$ are abstract elements in $\mathcal{D}_s^\sharp$, we design the domain operations for domain $\mathcal{D}_s^\sharp$ based on their corresponding operations in domain $\mathcal{D}_t^\sharp$, as shown in Fig. 5. The abstraction of each lattice operation, including meet, join, and inclusion testing, is defined by applying the corresponding abstractions of $\mathcal{D}_t^\sharp$ on each identical hemisphere. For each arithmetic and bit-vector binary operation bop, the intuitive abstraction bop_s can be defined by applying bop_t on every two hemispheres.

Moreover, we can make use of the concrete semantics and the signedness information of T_i and T_j' to reduce the amount of calling $split$ and $\sqcup_s$. For example, the unsigned division will always lead to a quotient in the 0-hemisphere, unless for $T_1 /_{ut} T_0'$. Therefore, we define the operation as: $\langle T_0,\ T_1\rangle /_{us} \left\langle T_0',\ T_1'\right\rangle \triangleq \left\langle T_0'',\ \bot_t\right\rangle \sqcup_s split(T_1 /_{ut} T_0')$, where $T_0'' \triangleq T_0 /_{ut} T_0' \sqcup_t T_0 /_{ut} T_1' \sqcup_t T_1 /_{ut} T_1'$. The

Type	Abstraction
Lattice	$\langle T_0, T_1\rangle \sqsubseteq_s \langle T_0', T_1'\rangle \triangleq T_0 \sqsubseteq_t T_0' \wedge T_1 \sqsubseteq_t T_1'$ $\langle T_0, T_1\rangle \sqcap_s \langle T_0', T_1'\rangle \triangleq \langle T_0 \sqcap_t T_0', T_1 \sqcap_t T_1'\rangle$ $\langle T_0, T_1\rangle \sqcup_s \langle T_0', T_1'\rangle \triangleq \langle T_0 \sqcup_t T_0', T_1 \sqcup_t T_1'\rangle$
Arithmetic and Bit-vector	$\langle T_0, T_1\rangle\ bop_s \langle T_0', T_1'\rangle \triangleq \sqcup_s\{R \mid R = split(T_i\ bop_t\ T_j'),\ i,j \in \{0,1\}\}$
Widening	$\langle T_0, T_1\rangle \triangledown_s \langle T_0', T_1'\rangle \triangleq \langle T_0 \sqcup_t T_0', T_1 \sqcup_t T_1'\rangle$

Fig. 5. Domain operations of the bit-level abstraction in swb

abstractions of other arithmetic and bit-vector operations are designed similarly and shown in the supplementary material.

Example 4.7. Given $P = \langle 0b00\mu 0, 0b1\mu\mu\mu\rangle, Q = \langle 0b00\mu 1, \bot_t\rangle$, we can observe that: when Q is 0b0001, then $0b1\mu\mu\mu /_{ut}\ 0b0001 \triangleq 0b1\mu\mu\mu$; when Q is $0b0011$, then the quotient must be in the 0-hemisphere. Therefore, $0b1\mu\mu\mu /_{ut}\ 0b00\mu 1$ may not be bottom in each hemisphere and needs to be split. In fact, for the unsigned division operation, the quotient can be in the 1-hemisphere only if the divisor is potentially 1 and the dividend is not bottom in the 1-hemisphere.

The widening operation of $\mathcal{D}_s^\sharp$ is usually replaced by its join operation. As mentioned in Sec. 4.1, we also design an efficient algorithm when the two abstract states have common trailing bits in the same hemisphere. In this case, we immediately set all other higher bits to be unknown but keep the sign bit unchanged.

Example 4.8. When analyzing the program in Fig. 3, with the above special consideration about widening operation, we have $\langle 0b0000u101, \bot_t\rangle \triangledown_s \langle 0b000uu101, \bot_t\rangle \triangleq \langle 0b0uuuu101, \bot_t\rangle$ at line 4.

Transfer Functions. The test transfer functions of domain $\mathcal{D}_s^\sharp$ are designed similarly to those in `tnum`$^+$ domain. For $P, Q \in \mathcal{D}_s^\sharp$, we compute the result P' of $P \leq_u Q$ as follows:

$$P' \triangleq \begin{cases} \bot_s & \text{if } Q = \bot_s \vee uminP > umaxQ \\ P & \text{if } Q.v + Q.m = umax \\ P \sqcap_s splitRange(P, Q) & \text{otherwise} \end{cases}$$

where $splitRange(P, Q) \triangleq split(tnum\_range\ (P.v,\ Q.v + Q.m))$, $uminP$ is the unsigned minimal abstract value of P, and $umaxQ$ is the unsigned maximal abstract value of Q.

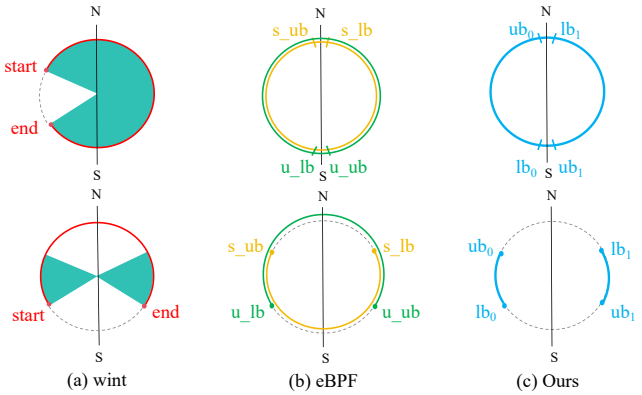
5 WORD-LEVEL ABSTRACTION

This section presents the word-level abstraction of swb, which tracks each hemisphere's lower and upper bounds. Compared to the wrapped interval domain [21, 47] (`wint`), it has two unique advantages: (1) It can express disjunctive properties depending on different signs; (2) It can be better integrated with the bit-level abstract domain since both track each hemisphere's values. Full details of the implementation are available in the supplementary material.

5.1 Domain Representation

Let $\bot_i$ represent an empty interval based on machine integer semantics. We use I_0 to denote either $\bot_i$ or a normal machine integer interval $(\!|lb_0, ub_0|\!)$ where $umin \leq lb_0 \leq ub_0 \leq smax$. Similarly, I_1 represents $\bot_i$ or a normal machine integer interval $(\!|lb_1, ub_1|\!)$ where $smin \leq lb_1 \leq ub_1 \leq umax$.

To represent the word-level abstraction of swb for a variable x, we use $\langle I_0, I_1\rangle$, which means that x belongs to either I_0 or I_1. The representation for an empty signed wrapped interval is $\bot_w$ or $\langle \bot_i, \bot_i\rangle$. On the other hand, $\top_w$ represents the interval $\left\langle (\!|umin, smax|\!), (\!|smin, umax|\!)\right\rangle$, which encompasses all possible values. We refer to $(\!|umin, smax|\!)$ as $\top_0$, and $(\!|smin, umax|\!)$ as $\top_1$.



(a) wint (b) eBPF (c) Ours

Fig. 6. Comparison between different interval-like abstractions.

The Galois connection between the concrete domain of powerset of of n-width bit-vectors $\mathcal{B}_n$ and the word-level abstraction of `swb` can be defined as: $(\wp(\mathcal{B}_n), \subseteq) \underset{\alpha_w}{\overset{\gamma_w}{\leftrightarrows}} (\mathcal{D}_w^\sharp, \sqsubseteq_w)$, where $\mathcal{D}_w^\sharp$ is the set of all signed wrapped interval over $\mathcal{B}_n$: $\{\langle (\!|lb_0, ub_0|\!), (\!|lb_1, ub_1|\!)\rangle \mid lb_0, ub_0, lb_1, ub_1 \in \mathcal{B}_n,\ umin \le lb_0 \le ub_0 \le smax, smin \le lb_1 \le ub_1 \le umax\}$. $\mathcal{D}_w^\sharp$ forms a complete lattice $(\mathcal{D}_w^\sharp, \sqsubseteq_w, \sqcap_w, \sqcup_w, \bot_w, \top_w)$. The abstraction function $\alpha_w\colon\ \wp(\mathcal{B}_n) \to \mathcal{D}_w^\sharp$ is:

$$\alpha_w(B) \triangleq \begin{cases} \bot_w & \text{if } B = \varnothing \\ \langle (\!|inf\ B, sup\ B|\!),\ \bot_t\rangle & \text{if } \forall b \in B, umin \le b \le smax \\ \langle \bot_t,\ (\!|inf\ B, sup\ B|\!)\rangle & \text{if } \forall b \in B, smin \le b \le umax \\ \langle (\!|inf\ B, sup\ B_0|\!),\ (\!|inf\ B_1, sup\ B|\!)\rangle & \text{otherwise} \end{cases}$$

where $B \subseteq \mathcal{B}_n$ is a subset of n-width bit-vectors, and $B_0 \triangleq \{b \in B \mid umin \le b \le smax\}$ while $B_1 \triangleq \{b \in B \mid smin \le b \le umax\}$.

The concretization function $\gamma_w : \mathcal{D}_w^\sharp \to \wp(\mathcal{B}_n)$ is defined as:

$$\gamma_w(P) \triangleq \begin{cases} \varnothing & \text{if } P = \bot_w \\ \{x \in \mathcal{B}_n \mid l \le x \le u\} & \text{if } P = \langle \bot_i,\ (\!|l, u|\!)\rangle \text{ or } P = \langle (\!|l, u|\!),\ \bot_i\rangle \\ \{x \in \mathcal{B}_n \mid lb_0 \le x \le ub_0 \vee lb_1 \le x \le ub_1\} & \text{if } P = \langle (\!|lb_0, ub_0|\!), (\!|lb_1, ub_1|\!)\rangle \end{cases}$$

where $P \in \mathcal{D}_w^\sharp$ represents an abstract element in the word-level abstraction of `swb`.

It is important to note that this abstract representation differs from the representation of the `wint` domain [21, 47], which uses $(\!|start, end|\!)$ to represent all values from $start$ to end clockwise. Meanwhile, `wint` allows $end < start$, by which it can represent all possible values in $(\!|umin, end|\!) \vee (\!|start, umax|\!)$. This can be more precise than our word-level domain, only when $end < start$ and they have the same sign. However, `wint` is less precise when representing two intervals in different signs. On the other hand, our word-level abstraction is always equal to the combination of signed and unsigned interval abstractions implemented in the kernel, which can be represented as $(\!|s\_lb, s\_ub|\!)$ and $(\!|u\_lb, u\_ub|\!)$ respectively.

The comparison is illustrated in Fig. 6, where the top (bottom) three circles represent a better (worse) analysis precision for `wint` domain. The arcs of those parts filled in the two leftmost circles of Fig. 6(a) represent the concrete values that we need to abstract in each case. Solid red and blue arcs represent `wint` and our word-level abstraction correspondingly; grass-green and yellow arcs represent the unsigned interval and signed interval abstraction of eBPF, respectively.

Type	Abstraction
Lattice	$\langle I_0,\ I_1\rangle \sqsubseteq_w \langle I'_0,\ I'_1\rangle \triangleq I_0 \sqsubseteq_i I'_0 \wedge I_1 \sqsubseteq_i I'_1$ $\langle I_0,\ I_1\rangle \sqcap_w \langle I'_0,\ I'_1\rangle \triangleq \langle I_0 \sqcap_i I'_0,\ I_1 \sqcap_i I'_1\rangle$ $\langle I_0,\ I_1\rangle \sqcup_w \langle I'_0,\ I'_1\rangle \triangleq \langle I_0 \sqcup_i I'_0,\ I_1 \sqcup_i I'_1\rangle$
Arithmetic and Bit-vector	$\langle I_0,\ I_1\rangle\ bop_w \langle I'_0,\ I'_1\rangle \triangleq \sqcup_w\{R \mid R = break(I_a\ bop_i\ I'_b),\ a,b \in \{0,1\}\}$
Widening	$\langle I_0,\ I_1\rangle \triangledown_w \langle I'_0,\ I'_1\rangle \triangleq \langle I_0 \triangledown_i I'_0,\ I_1 \triangledown_i I'_1\rangle$

Fig. 7. Domain operations of the word-level abstraction in swb

We can derive a signed wrapped interval from a non-bottom wrapped interval $W = (\!|start, end|\!)$ by the following auxiliary function:

$$break(W) \triangleq \begin{cases} \langle (\!|start, end|\!),\ \bot_i\rangle & \text{if } start \le end \le smax \\ \langle \bot_i,\ (\!|start, end|\!)\rangle & \text{if } smin \le start \le end \\ \langle (\!|start, smax|\!),\ (\!|smin, end|\!)\rangle & \text{if } start \le smax \wedge smin \le end \\ \langle (\!|umin, end|\!),\ (\!|start, umax|\!)\rangle & \text{if } end \le smax \wedge smin \le start \\ \top_w & \text{otherwise} \end{cases}$$

5.2 Domain Operators and Transfer Functions

Assuming $\langle I_0, I_1\rangle = \langle (\!|lb_0, ub_0|\!), (\!|lb_1, ub_1|\!)\rangle$ and $\langle I'_0, I'_1\rangle = \langle (\!|lb'_0, ub'_0|\!), (\!|lb'_1, ub'_1|\!)\rangle$ are abstract elements in $\mathcal{D}^\sharp_w$, we design the domain operations for the domain $\mathcal{D}^\sharp_w$ in the way similar to the bit-level abstraction $\mathcal{D}^\sharp_s$, as shown in Fig. 7. The sub-abstractions $\sqsubseteq_i$, $\sqcap_i$, and $\sqcup_i$ are defined like those in the classical interval domain [8], while bop_i are designed following those in `wint`.

The difference between the domain operations in our word-level abstraction $\mathcal{D}^\sharp_w$ with those in `wint` domain is: we split the machine integer circle into two hemispheres and always do operations between each of them, while `wint` chooses not to split the circle or just split the circle into one or two hemispheres depending on the operation as needed. For bit-vector operations, we only design abstractions for truncation and extension, while leaving other operations to simply return $\top_w$, similar to the implementation in the `wint` domain of the Crab library.

Meanwhile, our addition, subtraction, and multiplication operations are more precise than those in the kernel, as the kernel's interval operations return a top value upon detecting overflow. Additionally, the Linux kernel lacks division and remainder operations.

For n-bits integers $\mathcal{B}_n$, the lattice height of $\mathcal{D}^\sharp_w$ is 2^n. Although it is finite, we still need to design a widening operation to accelerate the fixed point iteration, as shown in Fig. 7, where $I_k \triangledown_i I'_k$ do widening by doubling the size of the interval like `wint` [21], but just consider the 0 or 1-hemisphere, rather than the whole circle.

The test transfer functions is designed similarly to the $\mathcal{D}^\sharp_s$ and $\mathcal{D}^\sharp_t$ domain. Here, we also take $P \le_u Q$ as an example, where $P, Q \in \mathcal{D}^\sharp_w$. If $Q = \bot_w$, then the result P' corresponding to P is obviously also $\bot_w$. Otherwise, for $Q = \langle I'_0, I'_1\rangle$, with $I'_1 = (\!|lb'_1, ub'_1|\!)$ and $I'_0 = (\!|lb'_0, ub'_0|\!)$, we compute the result P' as:

$$P' \triangleq \begin{cases} P \sqcap_w break((\!|umin, ub'_0|\!)) & \text{if } I'_1 = \bot_i \\ P \sqcap_w break((\!|umin, ub'_1|\!)) & \text{otherwise} \end{cases}$$

Note that we determine the intersection part from extreme value *umin*, since this can still retain enough analysis precision at the word level. The computation for Q' and the implementation of the signed comparison can be designed similarly.

6 COMBINING BIT-LEVEL AND WORD-LEVEL ABSTRACTIONS

Our analysis leverages a combination of word-level and bit-level abstractions by applying the reduced product operation, resulting in a newly defined domain, denoted as swb. Formally, this domain is expressed as $\mathcal{D}^\sharp_{swb} \triangleq \mathcal{D}^\sharp_s \times \mathcal{D}^\sharp_w$. Compared to the $\texttt{tnum}^+$ and wint domains, this product domain offers significantly higher precision, as it preserves disjunctive information regarding signs and integrates both bit-level and word-level abstractions effectively.

6.1 Reduced Product Operator Design

Inspired from the reduce operation, i.e., the *reg_bounds_sync* function [61] used in the kernel, the reduced product function ρ we design in this paper acts upon each hemisphere k ($k \in \{0, 1\}$) of machine integers independently, in the following three steps:

First, we update the word-level abstraction I_k to I'_k using the information of the bit-level abstraction T_k, since this bit-level abstraction also reveals the minimal and maximal potential values:

$$I'_k.lb = T_k.v > I_k.lb \ ? \ T_k.v \ : \ I_k.lb$$
$$I'_k.ub = T_k.v + T_k.m < I_k.ub \ ? \ T_k.v + T_k.m \ : \ I_k.ub$$

Then, we improve the bit-level abstraction T_k to T'_k with the bit information getting from the updated word-level abstraction I'_k, by:

$$T'_k = T_k \sqcap_t \ tnum\_range(I'_k.lb, \ I'_k.ub)$$

Finally, we update the bound of the word-level abstraction again from T'_k, in the way like the first step, by:

$$I''_k.lb = T'_k.v > I'_k.lb \ ? \ T'_k.v \ : \ I'_k.lb$$
$$I''_k.ub = T'_k.v + T'_k.m < I'_k.ub \ ? \ T'_k.v + T'_k.m \ : \ I'_k.ub$$

This is because the updated T'_k may also have a chance to improve the value bounds. We will prove that these three steps are sufficient to realize the standard reduced product operator in the next subsection.

Example 6.1. Suppose after the first step, $I'_k = (\!|0b0000, \ 0b0011|\!)$ and $T_k = (0b0000, \ 0b0110)$. Then, after the second step, we will refine the bit-level abstraction to $T'_k = (0b0000, \ 0b0010)$, which can also improve the word-level abstraction to $I''_k = (\!|0b0000, \ 0b0010|\!)$.

Compared to the abstraction reduction implemented in the kernel, our reduced product function does not need to transfer information between signed and unsigned intervals, thanks to the design of our word-level abstraction. Meanwhile, with the help of the standard reduced product operation, there are no need of *Abstraction/Reduction Operators* [61] in the bit-level and word-level abstraction. All abstract domains are designed independently, and they have independent algorithms and implementations for abstracting each operation, without relying on the abstract results of other abstract domains as inputs for computation.

It is also interesting to note that, like [61], we use the conventional terminology "reduced product" to denote the combination operator between the bit-level and word-level abstraction. However, The $\texttt{tnum}^+$ domain is non-convex, resembling set-based methods in some aspects but distinguished by its unique value encoding and mask mechanism. Therefore, the combination operator is also similar to the "Witness operator" [25], but still has some differences.

6.2 Soundness and Optimality

Recall the preliminary knowledge of reduced product operation in Sec. 2. Soundness of the operator means all concrete values should be contained in the abstraction of each component domain. After

that, the optimality of the operator means that the precision of each component abstraction can not be improved anymore, though with the help of another component abstraction.

Our reduced operation ρ adheres to soundness and optimality. Specifically, ρ ensures the construction of a sound swb abstraction while maintaining the optimality of its component abstractions. Soundness is straightforward to establish, as the three steps of ρ and all abstract operations of both bit-level and word-level abstractions are sound by construction. On the other hand, recent work by Vishwanathan et al. [61] has verified the soundness of similar *Abstraction/Reduction Operators*, which closely resemble our reduced function ρ. However, their work does not address optimality. In contrast, we provide a formal proof of the optimality of ρ, i.e, the bit-level and word-level abstractions cannot be improved anymore by their cooperation after doing ρ. Our proof assumes that ρ updates all lower and upper bounds in the first and third steps, representing the most complex cases. The proof for other cases follows a similar reasoning process.

Theorem 6.2. $tnum\_range(I_k''.lb,\ I_k''.ub) == tnum\_range(I_k'.lb,\ I_k'.ub)$.

Proof. From the above assumption, ρ updates all lower and upper bounds in the first and third steps, thus $I_k''.lb = T_k'.v$, $I_k''.ub = T_k'.v + T_k'.m$, $I_k'.lb = T_k.v$, and $I_k'.ub = T_k.v + T_k.m$. Meanwhile, suppose $tmp = tnum\_range(I_k'.lb,\ I_k'.ub) = tnum\_range(T_k.v,\ T_k.v + T_k.m)$, then from the design of meet operation, we have $T_k'.v = T_k.v \mid tmp.v$ and $T_k'.m = T_k.m \ \&\ tmp.m$ for the second step of ρ. So, $tnum\_range(I_k''.lb,\ I_k''.ub) = tnum\_range(T_k.v \mid tmp.v,\ T_k.v \mid tmp.v + T_k.m \ \&\ tmp.m)$. Recall the computation process of $tnum\_range$ in Sec. 2, $tmp.m$ sets all bits from the highest bit (denoted as l) of $T_k.m$ that is 1 to the lowest bit to be 1, while $tmp.v$ just retains those bits of $T_k.v$ upper than l. From this, we can know that $T_k.m \ \&\ tmp.m$ is equal to $T_k.m$, and $T_k.v \mid tmp.v$ is equal to $T_k.v$ also. Therefore, $tnum\_range(I_k''.lb,\ I_k''.ub)$ is equal to $tnum\_range(T_k.v,\ T_k.v + T_k.m)$, and the proof is established.

Theorem 6.3. *Reduced operation ρ finds optimal bit-level and word-level abstractions.*

Proof. The reduced operation ρ uses three steps to cooperate information between bit-level and word-level abstractions. Note that the third step is similar to the first step to improve the word-level abstraction with the help of bit-level abstraction. Also, we can only improve the bit-level abstraction similarly to the second step. Therefore, ρ finds the optimal abstractions after the above three steps, implying that there should not be another step that improves the bit-level abstraction with the help of word-level abstraction in a way like the second step. i.e., $T_k' == T_k' \sqcap_t tnum\_range(I_k''.lb,\ I_k''.ub)$. From Theorem 6.2, we only need to prove $T_k' == T_k' \sqcap_t tnum\_range(I_k'.lb,\ I_k'.ub)$. Because in the second step of ρ, $T_k' = T_k \sqcap_t tnum\_range(I_k'.lb,\ I_k'.ub)$, the proof is established.

Complexity. The word-level and bit-level domains proposed in this paper operate independently on each program variable. Consequently, the time complexity of both domains, as well as their combination (swb), is $O(n)$, where n represents the number of program variables. This ensures that the analysis scales efficiently to real-world programs.

7 EVALUATION

We implement the tnum$^+$ domain and the combined domain swb in the Crab library[1], and do some experiments to evaluate them. We aimed to answer three research questions:

- **RQ1**: How do the above domains perform when analyzing real-world programs with complex loop structures without bit-vector operations?
- **RQ2**: Can the combined domain aid in analyzing programs with intensive bit-vector operations from hardware verification?

[1]https://github.com/seahorn/crab

Table 3. Results for loop programs

Source	Files	LoC	wint		tnum⁺		swb	
			CR	Time	CR	Time	CR	Time
loops	32	652	20	0.55	20	0.63	20	1.36
loop-acceleration	26	412	17	0.54	16	0.59	**18**	1.13
loop-crafted	3	54	1	0.05	1	0.1	1	0.22
loop-invgen	25	5881	38	2.28	19	10.58	**45**	32.98
loop-lit	29	2081	19	0.7	4	0.74	**21**	2.39
loop-new	11	312	1	0.23	0	0.58	**2**	0.77
loop-industry-pattern	5	421	1	0.2	1	0.36	1	1.39
loops-crafted-1	51	1399	11	35.71	11	204.52	**16**	107.87
loop-invariants	7	127	2	0.13	2	0.16	**4**	0.25
loop-simple	8	127	7	19.19	1	17.36	7	42.36
loop-zilu	52	839	32	0.86	30	0.86	32	1.69

- **RQ3**: How does the combined domain behave in eBPF program analysis?

To answer RQ1 and RQ2, we evaluated a set of C programs using the Clam analyzer[2], an LLVM (version 14.0) frontend for Crab. We only use Clam's default parameters, such as the interprocedural analysis methods, widening thresholds, etc. The benchmarks for RQ1 are taken from the category "ReachSafety-Loops" of SV-COMP 2024[3], which is often used to evaluate the performance of different static analysis tools. Meanwhile, these benchmarks exclude bit-vector operations and thus are useful to evaluate how program analysis can be improved with the help of bit-level information. We only reserve those programs that can be analyzed by `wint` domain implemented in Crab. For RQ2, we used programs from the "BV" category of Word-level Hardware-Model-Checking Benchmarks[4] that consist of Btor2 circuits and their corresponding C programs translated by Btor2C [1] in lazy mode. These C programs consist of lots of bit-vector operations because of the translation. To answer RQ3, we implement `swb` domain in the eBPF verifier PREVAIL of version [23][5] that is implemented based on Crab also. Correspondingly, we use the benchmarks[6] that are included in the tool and consist of 208 programs that can pass the eBPF verifier in Linux.

All the experiments were performed on a machine running Ubuntu 22.04 (64-bit), with 256GB RAM and a 3.7 GHz 32-core AMD 3970X CPU. We set the time limit for analyzing each program of each question to 180 seconds.

7.1 RQ1. Analyzing Loop Programs without Bit-Vector Operations

The analysis results for SV-COMP loop programs are shown in Table 3, where "CR" denotes the count of assertions that Clam validates correctly. "Files" are the counts of tested programs for different sources of loop programs, and "LoC" represents the total lines of these programs, excluding comments and blanks. Note that, each program has at least one assertion. We record the analysis time of different domains in seconds. We only compare our `swb` domain and `tnum`⁺ domain with the `wint` domain, ignoring the other abstract domains because all the domains we design are not relational, and `wint` is the only widely used domain based on machine integer semantics.

Our findings indicate that bit-level abstraction `tnum`⁺ cannot validate most programs independently. This is normal because the benchmark programs only consist of arithmetic operations, and

[2]https://github.com/seahorn/clam

[3]https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/svcomp24-final?ref_type=tags

[4]https://gitlab.com/sosy-lab/research/data/word-level-hwmc-benchmarks/-/tree/tacas23-camera?ref_type=tags

[5]https://github.com/vbpf/ebpf-verifier/tree/d29fd26345c3126bf166cf1c45233a9b2f9fb0a0

[6]https://github.com/vbpf/ebpf-samples/tree/8307b929b2db298622a1e380b8610d5eebcdca32

the domain is mainly designed to retain bit-level information. However, when combining bit-level abstraction and word-level abstractions, our swb domain can validate 167 programs in total, much more than the existing wint domain that only validates 149 programs since it only focuses on word-level abstraction. This comparison reveals that it is also of great significance to retain both bit-level and word-level abstractions at the same time for arithmetic operations, even though the benchmark programs do not consist of bit-vector operations. Note that all these abstract domains are non-relational. Thus, it is not easy for them to validate relational assertions, not to mention those programs that are even difficult for relational abstract domains.

As for efficiency, analysis with $\texttt{tnum}^+$ domain typically spends time similarly to those with wint domain because they have the same complexity. However, $\texttt{tnum}^+$ sometimes may be much more inefficient, e.g., for "loop-invgen" and "loop-crafted-1", since we just replace the widening operation of $\texttt{tnum}^+$ with the join operation except for a special consideration of one useful common case. In the worst case, $\texttt{tnum}^+$ needs the same number of times as the bit-width to converge. Therefore, designing a more effective widening operation is of great significance in future work. Meanwhile, swb usually has to spend more time than the sum of $\texttt{tnum}^+$ and wint. This is obvious since swb combines two abstractions and retains information on different signs rather than $\texttt{tnum}^+$ and wint. However, swb spends less time than $\texttt{tnum}^+$ for "loop-crafted-1". This is because the efficient widening operation of the word-level abstraction in swb can alleviate the inefficiency of the bit-level abstraction and make swb converge faster.

In summary, when analyzing real-world programs with complex loop structures without bit-vector operations, our swb domain is much more precise than the existing wint domain, thanks to the combination of the bit-level and word-level abstraction. Meanwhile, swb also has considerable efficiency, though it has a more complicated design.

7.2 RQ2. Analyzing Hardware-Related Programs with Intensive Bit-Vector Operations

Table 4 introduces the benchmarks from the hardware verification, where "Source" represents the project they come from and is denoted with an "ID". "Files" is the total count of each project, while "LoC" means the total lines of the C programs translated from Verilog in lazy mode, excluding comments and blanks.

Table 5 shows the total experiment results of the comparison between Clam equipped with swb, CPAchecker[7] [2], and AVR[8] [24]. CPAchecker is the state-of-the-art software verifier based on model checking. AVR is the state-of-the-art hardware model checker that won Hardware Model Checking Competitions (HWMCC) 2020, and directly takes Btor2 as input, which is the modeling language used for HWMCC. AVR has implemented bounded model checking (BMC) [3], property directed reachability (PDR) [20], and k-induction [63]. "Time" is the total execution time in seconds of these tools. We only use the default parameters of these two tools, except for the same time limit as Clam. For each source of benchmarks, bold in Table 5 indicates that the number of correct validations is the highest among all comparison tools, and the red font indicates that the time required to validate is the least.

As we can see, AVR has correct validation results (CR) for most assertions in total, but spends lots of time. This is because AVR is operating the Btor2 circuit directly and has specially designed analysis algorithms, while other tools analyze more complex C programs. When using classical software verifiers like CPAchecker to validate the programs translated from hardware, we can not achieve the same performance as those designed for hardware verification, e.g., AVR. Among the three tools, CPAchecker validates the least assertions, nearly 28% of AVR, but spends the most

[7]https://cpachecker.sosy-lab.org/, version 2.3.1-svn
[8]https://github.com/aman-goel/avr, version 2.1

Table 4. Benchmarks from hardware verification

ID	Source	Files	LoC	ID	Source	Files	Loc
1	beem	664	5,041,059	7	mann-2020	1	102
2	btor2tools-examples	8	425	8	mann-data-integrity	76	270,064
3	goel-crafted	24	8,764	9	v2c-hwmcc15	390	657,652
4	goel-industry	241	4,325,173	10	wolf-2018D	280	63,368
5	goel-opensource	137	450,424	11	wolf-2019A	390	316,113
6	mann-2019	4	2,241	12	wolf-2019C	113	718,287

Table 5. Results for programs from hardware verification. Bold fonts mean the maximal assertions validated, while red fonts indicate minimal time spent.

ID	AVR		CPAchecker		Clam+swb	
	CR	Time(s)	CR	Time(s)	CR	Time(s)
1	212	87008.8	55	105897.7	**313**	1409.7
2	8	0.5	8	127.9	**9**	0.1
3	**22**	71.5	19	1737.5	8	1.4
4	**216**	6328.7	14	24299.8	5	701.9
5	**125**	2007.3	95	11083.9	44	26.6
6	1	539.8	1	465.2	**2**	1.2
7	**1**	0.1	0	180	0	0.1
8	15	12090.3	0	12341	**76**	47.3
9	**6**	2174.4	0	2528.3	0	58.4
10	**8**	977.1	0	2363.8	0	14.2
11	0	3058.1	0	1153.8	**17**	64.4
12	**63**	10302.7	2	18360.2	0	169.6
Total	**677**	124559.3	194	180539.1	474	2494.9

time, even 1.4 times more than AVR. Since CPAchecker is mainly based on model checking, it has to consume much more time. This is obvious, but the comparison between AVR and CPAchecker also shows that it is difficult for conventional software verifiers to validate bit-vector intensive C programs that are translated from hardware.

Fortunately, when equipped with swb domain, Clam validates more assertions for some sorts of benchmarks than AVR and CPAchecker. The total counts of Clam exceed 2.4 times more than CPAchecker while reaching 70.0% of AVR. In fact, because CPAchecker and AVR are based on model checking, they have substantial time consumption in constraint solving, leading to lots of assertions can not be validated. However, with the combination of bit-level and word-level abstraction, Clam is able to analyze the intensive bit-vector operations in these benchmarks with considerable precision. More surprisingly, thanks to the abstract interpretation framework, Clam spends the least total time among these tools, even only 2.0% of AVR and 1.3% of CPAchecker.

In summary, when analyzing programs with intensive bit-vector operations, our swb domain has great efficiency while still having considerable efficiency compared to state-of-the-art software and hardware verifiers, thus it is useful in aiding hardware verification.

7.3 RQ3. Analyzing eBPF Programs

We use our swb domain and wint domain to do eBPF verification and compare their performance because they are both based on machine integer semantics rather than mathematical semantics. For precision, our swb domain can verify 157 programs in total, much more than wint domain which can only verify 48 programs. All programs verified by wint can also be verified by swb. We

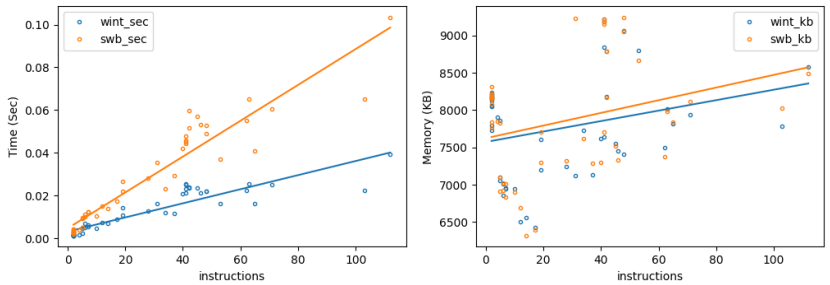


Fig. 8. Time (Sec) and memory usages (KB) of eBPF program analysis

notice that `wint` can only analyze these verified programs while meeting problems when analyzing the remaining programs. This is surprising, as it is a commonly used abstract domain in Crab. Meanwhile, the Linux eBPF verifier designs abstractions based on machine integers. From this perspective, our `swb` domain may play an important part when designing an eBPF verifier outside Linux in the future.

Fig. 8 shows the execution time in seconds and memory usage for analyzing eBPF programs with different numbers of instructions. `swb` almost spends twice as much time as the `wint`. Considering eBPF programs have no loop, this result is acceptable, because `swb` combines both word-level and bit-level abstractions, and is also signedness-aware. Fortunately, `swb` can verify these programs in very little time. Meanwhile, `swb` domain only consumes similar memory compared to `wint` domain.

In summary, when analyzing eBPF programs, our `swb` domain demonstrates higher precision compared to the `wint` domain, with similar memory usage, while also maintaining negligible analysis time.

8 DISCUSSIONS

Threats to Validity. We only implement the `swb` domain in the Crab library but pay little attention to the other components of a static analyzer since the abstract domain is the core of the abstract interpretation framework and has the potential to be included in other static analyzers. However, to adapt a static analyzer which is based on mathematical integer semantics to machine integer semantics, all other components should also be modified a lot, which is not easy. For example, Clam replaces unsigned comparisons with signed comparisons in a specific form when translating LLVM IR to its intermediate representation CrabIR. This process is sound in mathematical integer semantics but is not sound in machine integer semantics. Meanwhile, the assertion-checking component of Clam may also affect the final results, leading to false positives, though the program invariants here imply assertions to be true in fact.

Other Implications. One improvement of the eBPF program analysis based on `swb` domain is to characterize the disjunctive properties of programs, like what the range analysis in Linux kernel does. For instance, we can combine `swb` with trace partitioning [38] or design more precise disjunctive abstract domain [22, 27] based on `swb`. In this way, we may explore bugs in the eBPF verifier for both Windows [19] and Linux through differential testing, similar to existing compiler testing methods [37, 53]. Additionally, we can utilize the `swb` domain to assist in solving BV theory SMT problems [65], particularly those that involve a combination of bit-vector and arithmetic operations, which present a challenging issue [36, 64].

9 RELATED WORK

Abstract Domains Based on Machine Integer Semantics. A growing interest has been in redesigning word-level abstract domains to better model low-level code behaviors using machine

integer semantics [34, 35, 45, 46, 52, 54, 55]. Our word-level abstraction is closely related to the wrapped interval [21, 47], which improves the interval domain to track the effects of integer overflow. This abstraction is efficient, similar to the wrapped interval domain, and has higher accuracy in tracking values in different signs. Other means to determine the bit-precise interval abstraction, such as SAT [5, 26], BDD [50], quantifier elimination [33], and constrained optimization [66], may be more precise but are much more time-consuming.

Apart from the eBPF Verifier, word-level and bit-level abstractions are implemented in LLVM [59]. Regehr and Duongsaa [49] proposed the unsigned interval domain and bitwise domain, whose arithmetic operations are much slower than those of `tnum`. Vishwanathan et al. [60] improved the multiplication operation of `tnum`. To supplement existing domains in the Astrée [4], Miné [41] extended the classical interval domain with modular components and a bitfield domain focused only on bit-vector operation. Recently, Yoon et al. [67] improved loop-free program synthesis by combining unsigned interval, signed interval, and bitwise abstractions. Compared to the above works, our combination uses the standard reduced product operation for more generalized and precise domains and can analyze programs with loops.

Linux eBPF System Safety. Recently, the numerical range analysis implementation in the eBPF verifier has been verified to comply with the relevant soundness specifications [61]. In contrast, Sun and Su [56] utilized a technique called state embedding to identify logic bugs in the complete eBPF verifier, but this was done without carefully designed specifications. Additionally, Jitterbug [48] employs formal methods to verify the correctness of the eBPF JIT compiler across different architectures. Beyond formal verification, fuzz testing is also conducted to enhance the safety of eBPF [30, 51, 62]. BVF [57] generates structured eBPF programs that successfully pass the verifier. It then utilizes memory sanitation and kernel mechanisms to identify correctness bugs within the verifier. To uncover vulnerabilities in the eBPF runtime, BRF [29] improves program generation to adhere to the semantics and dependencies required by both the verifier and the eBPF subsystem.

10 CONCLUSION

We present an efficient and practical analysis for machine integers, combining new bit-level and word-level abstractions via standard reduced product operation. The bit-level abstraction generalizes the `tnum` abstraction in the eBPF verifier and extends it to track known bits more precisely in different signs. The word-level abstraction is a signed wrapped interval that can track lower and upper bounds in different signs, improving collaboration with the bit-level abstraction. We demonstrate the promising ability of our technique in various practical scenarios.

11 DATA AVAILABILITY

All source code, experimental results and benchmark programs are publicly available at Zenodo.

Acknowledgments

We thank the ISSTA 2025 reviewers for their constructive feedback. This work is supported by the National Key R&D Program of China (No.2022YFA1005101) and the National Natural Science Foundation of China (Nos.62032024, 62302434 and U2341212)

References

[1] Dirk Beyer, Po-Chun Chien, and Nian-Ze Lee. 2023. Bridging hardware and software analysis with Btor2C: A word-level-circuit-to-C translator. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 152–172. doi:10.1007/978-3-031-30820-8_12

[2] Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *Computer Aided Verification*. Springer, Berlin, Heidelberg, 184–190. doi:10.1007/978-3-642-22110-1_16

[3] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Amsterdam, The Netherlands, March 22-28, 1999*. Springer. doi:10.1007/3-540-49059-0_14

[4] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. 196–207. doi:10.1145/781131.781153

[5] Jörg Brauer and Andy King. 2010. Automatic abstraction for intervals using Boolean formulae. In *Proceedings of the 17th International Conference on Static Analysis* (Perpignan, France) *(SAS'10)*. Springer, Berlin, Heidelberg, 167–183.

[6] Patrick Cousot. 2021. *Principles of abstract interpretation*. MIT Press.

[7] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 238–252. doi:10.1145/512950.512973

[8] Patrick Cousot and Radhia Cousot. 1977. Static determination of dynamic properties of generalized type unions. In *Proceedings of an ACM Conference on Language Design for Reliable Software* (Raleigh, North Carolina). ACM, New York, NY, USA, 77–94. doi:10.1145/800022.808314

[9] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Tucson, Arizona) *(POPL '78)*. ACM, New York, NY, USA, 84–96. doi:10.1145/512760.512770

[10] CWE-190. 2024. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html

[11] CWE-191. 2024. CWE-191: Integer Underflow (Wrap or Wraparound). https://cwe.mitre.org/data/definitions/191.html

[12] CWE-192. 2024. CWE-192: Integer Coercion Error. https://cwe.mitre.org/data/definitions/192.html

[13] CWE-194. 2024. CWE-194: Unexpected Sign Extension. https://cwe.mitre.org/data/definitions/194.html

[14] CWE-195. 2024. CWE-195: Signed to Unsigned Conversion Error. https://cwe.mitre.org/data/definitions/195.html

[15] CWE-196. 2024. CWE-196: Unsigned to Signed Conversion Error. https://cwe.mitre.org/data/definitions/196.html

[16] CWE-197. 2024. CWE-197: Numeric Truncation Error. https://cwe.mitre.org/data/definitions/197.html

[17] CWE-680. 2024. CWE-680: Integer Overflow to Buffer Overflow. https://cwe.mitre.org/data/definitions/197.html

[18] CWE-681. 2024. CWE-681: Incorrect Conversion between Numeric Types. https://cwe.mitre.org/data/definitions/681.html

[19] Poorna Gaddehosur Dave Thaler. 2021. Making eBPF work on Windows. https://opensource.microsoft.com/blog/2021/05/10/making-ebpf-work-on-windows/

[20] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. 2011. Efficient implementation of property directed reachability. In *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA*. FMCAD Inc., 125–134. http://dl.acm.org/citation.cfm?id=2157675

[21] Graeme Gange, Jorge A Navas, Peter Schachte, Harald Søndergaard, and Peter J Stuckey. 2015. Interval analysis and machine arithmetic: Why signedness ignorance is bliss. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 37, 1 (2015), 1–35. doi:10.1145/2651360

[22] Graeme Gange, Jorge A Navas, Peter Schachte, Harald Søndergaard, and Peter J Stuckey. 2021. Disjunctive interval analysis. In *International Static Analysis Symposium*. Springer, Berlin, Heidelberg, 144–165. doi:10.1007/978-3-030-88806-0_7

[23] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. 2019. Simple and precise static analysis of untrusted Linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. ACM, New York, NY, USA, 1069–1084. doi:10.1145/3314221.3314590

[24] Aman Goel and Karem Sakallah. 2020. AVR: Abstractly Verifying Reachability. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 413–422. doi:10.1007/978-3-030-45190-5_23

[25] Sumit Gulwani, Tal Lev-Ami, and Mooly Sagiv. 2009. A combination framework for tracking partition sizes. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*. ACM, 239–251. doi:10.1145/1480881.1480912

[26] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. 2008. Program analysis as constraint solving. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. ACM, New York, NY, USA, 281–292. doi:10.1145/1375581.1375616

[27] Arie Gurfinkel and Sagar Chaki. 2010. Boxes: A symbolic abstract domain of boxes. In *International Static Analysis Symposium*. Springer, Berlin, Heidelberg, 287–303. doi:10.1007/978-3-642-15769-1_18

[28] Arie Gurfinkel and Jorge A Navas. 2021. Abstract interpretation of LLVM with a region-based memory model. In *International Workshop on Numerical Software Verification*. Springer, 122–144. doi:10.1007/978-3-030-95561-8_8

[29] Hsin-Wei Hung and Ardalan Amiri Sani. 2024. BRF: Fuzzing the eBPF Runtime. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1152–1171. doi:10.1145/3643778

[30] Juan José López Jaimez and Meador Inge. 2023. Buzzer. https://github.com/google/buzzer

[31] Bertrand Jeannet and Antoine Miné. 2009. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Computer Aided Verification*. Springer, Berlin, Heidelberg, 661–667. doi:10.1007/978-3-642-02658-4_52

[32] Pankaj Kumar Kalita, Sujit Kumar Muduli, Loris D'Antoni, Thomas Reps, and Subhajit Roy. 2022. Synthesizing abstract transformers. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 1291–1319. doi:10.1145/3563334

[33] Deepak Kapur. 2006. Automatically generating loop invariants using quantifier elimination. Schloss-Dagstuhl-Leibniz Zentrum für Informatik.

[34] Andy King and Harald Søndergaard. 2008. Inferring congruence equations using SAT. In *Computer Aided Verification: 20th International Conference, CAV 2008 Princeton, NJ, USA, July 7-14, 2008 Proceedings 20*. Springer, 281–293. doi:10.1007/978-3-540-70545-1_2

[35] Andy King and Harald Søndergaard. 2010. Automatic abstraction for congruences. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 197–213. doi:10.1007/978-3-642-11319-2_16

[36] Jaehyung Lee and Woosuk Lee. 2023. Simplifying Mixed Boolean-Arithmetic Obfuscation by Program Synthesis and Term Rewriting. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. ACM, New York, NY, USA, 2351–2365. doi:10.1145/3576915.3623186

[37] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. NNSmith: Generating Diverse and Valid Test Cases for Deep Learning Compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. ACM, New York, NY, USA, 530–543. doi:10.1145/3575693.3575707

[38] Laurent Mauborgne and Xavier Rival. 2005. Trace Partitioning in Abstract Interpretation Based Static Analyzers. In *Programming Languages and Systems*. Springer, Berlin, Heidelberg, 5–20. doi:10.1007/978-3-540-31987-0_2

[39] Antoine Miné. 2001. A new numerical abstract domain based on difference-bound matrices. In *Programs as Data Objects: Second Symposium, PADO2001 Aarhus, Denmark, May 21–23, 2001 Proceedings*. Springer, 155–172. doi:10.1007/3-540-44978-7_10

[40] Antoine Miné. 2006. The octagon abstract domain. *Higher-order and symbolic computation* 19 (2006), 31–100. doi:10.1007/s10990-006-8609-1

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

[42] Antoine Miné et al. 2017. Tutorial on static inference of numeric invariants by abstract interpretation. *Foundations and Trends® in Programming Languages* 4, 3-4 (2017), 120–372.

[43] Rajdeep Mukherjee, Daniel Kroening, and Tom Melham. 2015. Hardware verification using software analyzers. In *2015 IEEE Computer Society Annual Symposium on VLSI*. IEEE, 7–12. doi:10.1109/ISVLSI.2015.107

[44] Rajdeep Mukherjee, Michael Tautschnig, and Daniel Kroening. 2016. v2c–A verilog to C translator. In *Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference, TACAS 2016, Eindhoven, The Netherlands, April 2-8, 2016*. Springer, 580–586. doi:10.1007/978-3-662-49674-9_38

[45] Markus Müller-Olm and Helmut Seidl. 2005. Analysis of modular arithmetic. In *Programming Languages and Systems: 14th European Symposium on Programming, ESOP 2005, Edinburgh, UK, April 4-8, 2005*. Springer, 46–60. doi:10.1007/978-3-540-31987-0_5

[46] Markus Müller-Olm and Helmut Seidl. 2007. Analysis of modular arithmetic. *ACM Trans. Program. Lang. Syst.* 29, 5 (Aug. 2007), 29–es. doi:10.1145/1275497.1275504

[47] Jorge A Navas, Peter Schachte, Harald Søndergaard, and Peter J Stuckey. 2012. Signedness-agnostic program analysis: Precise integer bounds for low-level code. In *Programming Languages and Systems: 10th Asian Symposium, APLAS 2012, Kyoto, Japan, December 11-13, 2012. Proceedings 10*. Springer, 115–130. doi:10.1007/978-3-642-35182-2_9

[48] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. 2020. Specification and verification in the field: Applying formal methods to {BPF} just-in-time compilers in the linux kernel. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 41–61.

[49] John Regehr and Usit Duongsaa. 2006. Deriving abstract transfer functions for analyzing embedded software. *ACM SIGPLAN Notices* 41, 7 (2006), 34–43. doi:10.1145/1159974.1134657

[50] John Regehr and Alastair Reid. 2004. HOIST: a system for automatically deriving static analyzers for embedded systems. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, MA, USA) *(ASPLOS XI)*. ACM, New York, NY, USA, 133–143. doi:10.1145/1024393.1024410

[51] Simon Scannell. 2020. eBPF Fuzzer. https://scannell.io/posts/ebpf-fuzzing

[52] Thomas Seed, Chris Coppins, Andy King, and Neil Evans. 2023. Polynomial analysis of modular arithmetic. In *International Static Analysis Symposium*. Springer, 508–539. doi:10.1007/978-3-031-44245-2_22

[53] Mayank Sharma, Pingshi Yu, and Alastair F. Donaldson. 2023. RustSmith: Random Differential Compiler Testing for Rust. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. ACM, New York, NY, USA, 1483–1486. doi:10.1145/3597926.3604919

[54] Tushar Sharma and Thomas Reps. 2017. Sound bit-precise numerical domains. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 500–520. doi:10.1007/978-3-319-52234-0_27

[55] Axel Simon and Andy King. 2007. Taming the wrapping of integer arithmetic. In *International Static Analysis Symposium*. Springer, 121–136. doi:10.1007/978-3-540-74061-2_8

[56] Hao Sun and Zhendong Su. 2024. Validating the {eBPF} verifier via state embedding. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 615–628.

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

[58] Joseph Tafese, Isabel Garcia-Contreras, and Arie Gurfinkel. 2023. BTOR2MLIR: A Format and Toolchain for Hardware Verification.. In *FMCAD*. 55–63.

[59] Jubi Taneja, Zhengyang Liu, and John Regehr. 2020. Testing static analyses for precision and soundness. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 81–93. doi:10.1145/3368826.3377927

[60] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2022. Sound, precise, and fast abstract interpretation with tristate numbers. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 254–265. doi:10.1109/CGO53902.2022.9741267

[61] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2023. Verifying the Verifier: eBPF Range Analysis Verification. In *International Conference on Computer Aided Verification*. Springer, 226–251. doi:10.1007/978-3-031-37709-9_12

[62] Dmitry Vyukov and Andrey Konovalov. 2015. Syzkaller: an unsupervised coverage-guided kernel fuzzer,. https://github.com/google/syzkaller

[63] Thomas Wahl. 2013. The k-induction principle. *Northeastern University, College of Computer and Information Science* (2013), 1–2.

[64] Dongpeng Xu, Binbin Liu, Weijie Feng, Jiang Ming, Qilong Zheng, Jing Li, and Qiaoyan Yu. 2021. Boosting SMT solver performance on mixed-bitwise-arithmetic expressions. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. ACM, New York, NY, USA, 651–664. doi:10.1145/3453483.3454068

[65] Peisen Yao, Qingkai Shi, Heqing Huang, and Charles Zhang. 2020. Fast bit-vector satisfiability. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 38–50. doi:10.1145/3395363.3397378

[66] Peisen Yao, Qingkai Shi, Heqing Huang, and Charles Zhang. 2021. Program analysis via efficient symbolic abstraction. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–32. doi:10.1145/3485495

[67] Yongho Yoon, Woosuk Lee, and Kwangkeun Yi. 2023. Inductive program synthesis via iterative forward-backward abstract interpretation. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1657–1681. doi:10.1145/3591288

Received 2024-10-27; accepted 2025-03-31