An Abstract Domain for Bit-Vector Inequalities*

Tushar Sharma[1], Aditya Thakur[1], and Thomas Reps[1,2]

[1] University of Wisconsin; Madison, WI, USA
[2] GrammaTech, Inc.; Ithaca, NY, USA

Abstract. This paper advances the state of the art in abstract interpretation of machine code. It tackles two of the biggest challenges in machine-code analysis: (1) holding onto invariants about values in memory, and (2) identifying affine-inequality invariants while handling overflow in arithmetic operations over bit-vector data-types.

Most current approaches either capture relations only among registers (and ignore memory entirely), or make potentially unsound assumptions when handling memory. Furthermore, existing bit-vector domains are able to represent either relational affine equalities or non-relational inequalities (e.g., intervals).

The key insight to tackling both challenges is to define a new domain combinator (denoted by $\mathcal{V}$), called the *view-product combinator*. $\mathcal{V}$ constructs a reduced product of two domains in which shared *view-variables* are used to communicate information among the domains. $\mathcal{V}$ applied to a non-relational memory domain and a relational bit-vector affine-equality domain constructs the *Bit-Vector Memory-Equality Domain* ($\mathcal{BVME}$), a domain of bit-vector affine-equalities over memory and registers. $\mathcal{V}$ applied to the $\mathcal{BVME}$ domain and a bit-vector interval domain constructs the *Bit-Vector Memory-Inequality Domain*, a domain of relational bit-vector affine-inequalities over memory and registers.

1 Introduction

The aim of the paper is to expand the set of techniques available for abstract interpretation and model checking of machine code [1, 24, 35]. It tackles two of the biggest challenges in machine-code analysis [30]: (1) holding onto invariants about values in memory, and (2) identifying affine-inequality invariants while handling overflow in arithmetic operations over bit-vector data-types.

Challenge 1: Memory-Value Invariants. When analyzing machine-code, memory is usually modeled as a flat array. When analyzing Intel x86 machine code, for instance, memory is modeled as a map from 32-bit bit-vectors to 8-bit bit-vectors. Consequently, an analysis has to deal with complications arising from the Little-Endian addressing mode and aliasing.

* Supported, in part, by NSF under grant CCF-0904371; by ONR under grants N00014-{09-1-0510, 11-C-0447}; by AFRL under contract FA8650-10-C-7088; and by DARPA under cooperative agreement HR0011-12-2-0012. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring agencies.

Example 1. Consider the following machine-code snippet:

```
mov eax, [ebp]
mov [ebp+2], ebx
```

The first instruction loads the four bytes pointed to by register **ebp** into the 32-bit register **eax**. Suppose that the value in register **ebp** is A. After the first instruction, the bytes of **eax** contain, in least-significant to most-significant order, the value at memory location A, the value at location $A+1$, the value at location $A+2$, and the value at location $A+3$. The second instruction stores the value in register **ebx** into the memory pointed to by **ebp**+2. Due to this instruction, the values at memory locations $A+2$ through $A+5$ are overwritten, after which the value in register **eax** no longer equals (the little-endian interpretation of) the bytes in memory pointed to by **ebp**. □

Challenge 2: Relational-Inequality Invariants over Bit-Vectors. Seminal work by Cousot and Halbwachs [10] defined the polyhedral domain, which is capable of expressing relational affine inequalities over rational (or real) variables. However, the native machine-integer data-types used in programs (e.g., `int`, `unsigned int`, `long`, etc.) perform bit-vector arithmetic, and arithmetic operations wrap around on overflow. Thus, the underlying point space used in the polyhedral domain does not faithfully model bit-vector arithmetic, and consequently the conclusions drawn from an analysis based on the polyhedral domain are unsound, unless special steps are taken [33].

Example 2. The following C-program fragment incorrectly computes the average of two `int`-valued variables [2]:

```
int low, high, mid;
assume (0 <= low <= high);
mid  = (low + high) / 2;
assert (0 <= low <= mid <= high);
```

A static analysis based on polyhedra would draw the unsound conclusion that the assertion always holds. In particular, assuming 32-bit `int`s, when the sum of low and high is greater than $2^{31}-1$, the sum overflows to a negative value, and the resulting value of mid is negative. Consequently, there exist runs in which the assertion fails. These runs are overlooked when the polyhedral domain is used for static analysis because the domain fails to take into account the bit-vector semantics of program variables. □

The problem that we wish to solve is *not* one of merely *detecting* overflow—e.g., to restrain an analyzer from having to explore what happens after an overflow occurs. On the contrary, our goal is to be able to track soundly the effects of arithmetic operations, including wrap-around effects of operations that overflow. This ability is useful, for instance, when analyzing code generated by production code generators, such as dSPACE TargetLink [11], which use the "compute-through-overflow" technique [15]. Furthermore, clever idioms for bit-twiddling

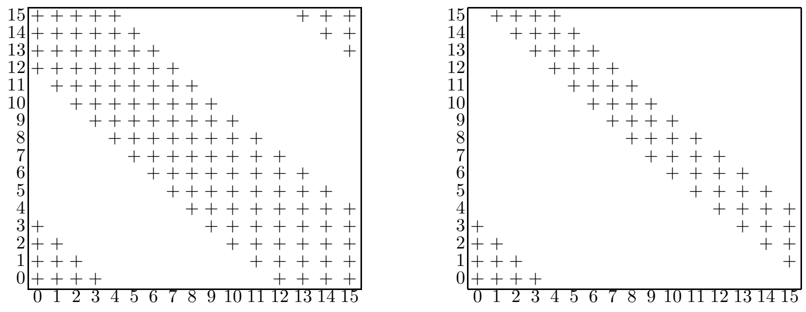
(a) $x + y + 4 \leq 7$ (b) $x + y \leq 3$

Fig. 1: Each + represents a solution of the indicated inequality in 4-bit unsigned modular arithmetic.

operations, such as the ones explained in Warren's book [37], sometimes rely on overflow.

The ideas used in designing an inequality domain for reals do not carry over to one designed for bit-vectors. First, in bit-vector arithmetic, positive constant factors cannot be canceled on both sides of an inequality; for example, if x and y are 4-bit bit-vectors, then $(4, 4)$ is in the solution set of $2x + 2y \leq 4$, but not of $x + y \leq 2$. Second, in bit-vector arithmetic, additive constants cannot be canceled on both sides of an inequality, as illustrated in the following example.

Example 3. Let x and y be 4-bit bit-vectors. Fig. 1(a) and Fig. 1(b) depict the solutions in bit-vector arithmetic of the inequalities $x+y+4 \leq 7$ and $x+y \leq 3$, respectively. Although $x+y+4 \leq 7$ and $x+y \leq 3$ are syntactically quite similar, their solution spaces are quite different. In particular, because of wrap-around of values computed on the left-hand sides using modular arithmetic, one cannot just subtract 4 from both sides to convert the inequality $x + y + 4 \leq 7$ into $x + y \leq 3$. □

Some techniques exist that are capable of representing certain kinds of inequalities over bit-vectors and handling memory soundly, e.g., the value-set analysis (VSA) used in CodeSurfer/x86 [1]. VSA uses a domain of intervals with a congruence constraint, sometimes called "strided-intervals" [29, 32]. However, VSA is an "independent-attribute analysis"—i.e., it is not capable of expressing relations among the values of multiple memory locations and registers.

Recent work has developed several abstract domains of *relational affine equalities* over variables that hold machine integers [27, 20, 13]. These domains *do* account for wrap-around on overflow. With respect to their analysis capabilities, the drawback of these domains is that they are unable to identify *inequality* invariants.

Problem Summary. These challenges lead to the following problem statement:

> Design an abstract domain of relational bit-vector affine-inequalities over memory-values and registers.

Key Insight: The View-Product Combinator. The key insight used to tackle both challenges involves a new domain combinator (denoted by $\mathcal{V}$), called the *view-product combinator*. $\mathcal{V}$ constructs a reduced product of two domains [8], using shared *view-variables* to communicate information between the domains. The following example illustrates the concept of a view-variable:

Example 4. Consider the equality constraint $H := x + 2\texttt{mm}[y + 2] = 4$, where x and y are bit-vector variables and $\texttt{mm}$ is the memory map. The term $\texttt{mm}[e]$ denotes the contents of $\texttt{mm}$ at address e. Let $\mathcal{E}$ denote any of the abstract domains of relational affine equalities over bit-vector variables [27, 20, 13]. H cannot be expressed using $\mathcal{E}$ alone. However, the formulas $x + 2\texttt{mm}[y + 2] = 4$ and $u = \texttt{mm}[y+2] \wedge x+2u = 4$ are equisatisfiable. The variable u is called a *view-variable*; the constraint $u = \texttt{mm}[y + 2]$ is the *view-constraint* associated with u.

The equality $x+2u = 4$ can be expressed using $\mathcal{E}$; therefore, what we require is a second abstract domain capable of expressing invariants that involve memory accesses, such as $u = \texttt{mm}[y + 2]$. This need motivates the *bit-vector memory domain* $\mathcal{M}$ that we introduce in §4. The constraint H can then be expressed by (i) introducing view-variable u, (ii) representing u's view-constraint in $\mathcal{M}$, (iii) extending $\mathcal{E}$ with u, and (iv) using $\mathcal{M}$ and $\mathcal{E}$ together. $\square$

The *Bit-Vector Memory-Equality Domain* $\mathcal{BVME}$, a domain of bit-vector affine-equalities over variables and memory-values, is created by applying the view-product combinator $\mathcal{V}$ to the bit-vector memory domain (§4) and the bit-vector equality domain. The *Bit-Vector Inequality Domain* $\mathcal{BVI}$, a domain of bit-vector affine-inequalities over variables, is created by applying $\mathcal{V}$ to the bit-vector equality domain and a bit-vector interval domain. The *Bit-Vector Memory-Inequality Domain* $\mathcal{BVMI}$, a domain of relational bit-vector affine-inequalities over variables and memory, is then created by applying $\mathcal{V}$ to the $\mathcal{BVME}$ domain and the bit-vector interval domain. The latter construction illustrates that $\mathcal{V}$ composes: the $\mathcal{BVMI}$ domain is created via two applications of $\mathcal{V}$.

The design of the view-product combinator $\mathcal{V}$ was inspired by the Subpolyhedra domain [23] (SubPoly), a domain for inferring relational linear inequalities over rationals. SubPoly is constructed as a reduced product of an affine-equality domain $\mathcal{K}$ over rationals [18], and an interval domain $\mathcal{J}$ over rationals [7] in which *slack variables* are used to communicate between the two domains. Such a design enables SubPoly to be as expressive as Polyhedra, but more scalable. $\mathcal{V}$ provides a generalization of the construction used in SubPoly. In fact, SubPoly can be constructed by applying $\mathcal{V}$ to $\mathcal{K}$ and $\mathcal{J}$ (see Eqn. (1) in §8).

Enabling Technology: Automatic Synthesis of Best Abstract Operations. Using a framework developed in prior work [34], we give a procedure for *synthesizing best abstract operations* for general reduced-product domains. In particular, we use this framework to ensure that the transformers for the reduced-product domains constructed via $\mathcal{V}$ are sound and precise, thereby guaranteeing that analysis results will be a conservative over-approximation of the concrete semantics.

```
    mov  ecx, [ebp−4]       //ecx = mm[ebp − 4]
    add  ecx, eax           //ecx = ecx + eax
    cmp  ecx, 10d           //if ecx >_u 10
    ja   L                  //     goto L
    xor  ecx, ecx           // ecx = 0
   0 ≤ mm[ebp − 4] + eax ≤ 10
L:
```

Fig. 2: Example snippet of Intel x86 machine code.

Contributions. The contributions of the paper are:

– The bit-vector memory domain, a non-relational memory domain capable of expressing invariants involving memory accesses (§4).
– The view-product combinator $\mathcal{V}$, a general procedure to construct more expressive domains (§5).
– Three domains for machine-code analysis constructed using $\mathcal{V}$:
  • The bit-vector memory-equality domain $\mathcal{BVME}$, which captures equality relations among bit-vector variables and memory (Defn. 6).
  • The bit-vector inequality domain $\mathcal{BVI}$, which captures inequality relations among bit-vector variables (Defn. 7).
  • The bit-vector memory-inequality domain $\mathcal{BVMI}$, which captures inequality relations among bit-vector variables and memory (Defn. 8).
– A procedure for synthesizing best abstract operations for reduced products of domains that meet certain requirements (§6).
– Experimental results that illustrate the effectiveness of the $\mathcal{BVI}$ domain applied to machine-code analysis (§7). On average (geometric mean), our $\mathcal{BVI}$-based analysis is about 3.5 times slower than an affine-equality-based analysis, while finding improved (more-precise) invariants at 29% of the branch points.

§2 provides an overview of our solution. §3 defines terminology. §8 describes related work. §9 concludes.

2 Overview

In this section, we illustrate the design of the bit-vector memory-inequality domain. Consider the Intel x86 machine-code snippet shown in Fig. 2. Pseudo-code for each instruction is shown at the right-hand side of each instruction. Note that the **ja** instruction, "jump if above, unsigned", treats **ecx** as unsigned. Thus, control reaches the **xor** instruction only if the value in **ecx** is greater than or equal to 0, and less than or equal to 10. Let $R := \{eax, ecx, ebp\}$ denote the set of register variables, and mm denote the memory map.

The highlighted text in Fig. 2 states the invariant $H := 0 \leq \mathtt{mm}[ebp{-}4]{+}eax \leq 10$ that holds after the **xor** instruction. Again, let $\mathcal{E}$ denote any of the abstract domains of relational affine equalities over bit-vector variables [27, 20, 13]. The invariant H cannot be represented using $\mathcal{E}$, because (i) H involves an invariant about a value in memory mm, and (ii) H is an inequality. To handle memory, we introduce the bit-vector memory domain $\mathcal{M}$, which is capable of expressing certain constraints on memory accesses. In particular, we can use $\mathcal{M}$ to express the constraint $u = \texttt{mm}[ebp - 4]$, where u is a fresh variable. We call u a *view-variable* and $u = \texttt{mm}[ebp - 4]$ a *view-constraint* for u. H can be written as the equisatisfiable formula $H_m := u = \texttt{mm}[ebp - 4] \wedge 0 \leq u + eax \leq 10$.

Notice there are no memory accesses in the constraint $0 \leq u + eax \leq 10$. The inequality $0 \leq u+eax \leq 10$ and $u+eax = s \wedge 0 \leq s \leq 10$ are equisatisfiable, where s is a fresh variable. Similar to u, s is a view-variable with $u + eax = s$ is a view-constraint for s. Thus, H_m can be rewritten as $H_{mi} := u = \texttt{mm}[ebp-4] \wedge u+eax = s \wedge 0 \leq s \leq 10$. Furthermore, $u + eax = s$ can be expressed in $\mathcal{E}$, and $0 \leq s \leq 10$ can be expressed using a bit-vector interval domain $\mathcal{I}$. Thus, by introducing the view-variables u and s, the invariant H can be expressed using $\mathcal{M}$, $\mathcal{E}$, and $\mathcal{I}$ together. The following derivation illustrates the above decomposition of the invariant H:

$$\textsc{Memory}\ \frac{u = \texttt{mm}[ebp - 4] \qquad \dfrac{u + eax = s \qquad s \in [0, 10]}{0 \leq u + eax \leq 10}\ \textsc{Inequality}}{0 \leq \texttt{mm}[ebp - 4] + eax \leq 10}$$

Note that the view-constraints $u = \texttt{mm}[ebp - 4]$ and $s = u + eax$ do not directly constrain the values of R and $\texttt{mm}$; they only constrain the view-variables u and s. The shared view-variables are used to exchange information among the various domains. In particular, the view-variable u is used to exchange information between memory domain $\mathcal{M}$ and equality domain $\mathcal{E}$, and the view-variable s is used to exchange information between $\mathcal{E}$ and interval domain $\mathcal{I}$.

3 Terminology

We assume that there is a Galois connection $\mathcal{G} = \mathcal{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A}$ between abstract domain $\mathcal{A}$ and concrete domain $\mathcal{C}$. For a given family of abstract domains $\mathcal{A}$, $\mathcal{A}[V]$ denotes the specific instance of $\mathcal{A}$ that is defined over vocabulary V, where V can contain function symbols and variables.

For a somewhat technical reason, we introduce the device of an *abstract-domain constructor*. Given an abstract-domain family $\mathcal{A}$ and vocabularies V_1 and V_2, an abstract-domain constructor for $\mathcal{A}$, denoted by $\mathfrak{C}_{\mathcal{A}}(V_1, V_2)$, constructs $\mathcal{A}[V_3]$, where $V_1 \subseteq V_3 \subseteq V_1 \uplus V_2$. In particular, $\mathfrak{C}_{\mathcal{A}}$ is free to decide what subset of V_2 to use when constructing $\mathcal{A}[V_3]$. (In our applications, the abstract-domain constructors either use all of V_2 or none of V_2.)

Let $A \in \mathcal{A}[V]$; we denote by $A \downarrow_{V_1}$ the value obtained by projecting A onto the vocabulary $V_1 \subseteq V$. We use $\oplus$ to denote a *vocabulary-extension operator* over domains; in particular, given domain $\mathcal{A}[V_1]$ and vocabulary V_2, $\mathcal{A}[V_1] \oplus V_2 = \mathcal{A}[V_1 \uplus V_2]$.

We now define the main concepts in *Symbolic Abstract Interpretation* [31] Let $\mathcal{L}$ be a logic, and $[\![\varphi]\!]$ denote the meaning of $\varphi \in \mathcal{L}$.

1. Given an abstract value $A \in \mathcal{A}[V]$, the *symbolic concretization* of A, denoted by $\widehat{\gamma}(A)$, maps A to a formula $\widehat{\gamma}(A)$ such that A and $\widehat{\gamma}(A)$ represent the same set of concrete states (i.e., $\gamma(A) = [\![\widehat{\gamma}(A)]\!]$).
2. Given $\varphi \in \mathcal{L}$, the symbolic abstraction of φ, denoted by $\widehat{\alpha}(\varphi)$, maps φ to the best value in $\mathcal{A}$ that over-approximates $[\![\varphi]\!]$ (i.e., $\widehat{\alpha}(\varphi) = \alpha([\![\varphi]\!])$).

Experience shows that for most abstract domains it is easy to write a $\widehat{\gamma}$ function (item 1) [31]. Several frameworks exist for computing $\widehat{\alpha}$ [31,36,34], including ones that apply to abstract domains whose elements are abstractions of transition relations (i.e., relations over assignments to pre-state/post-state variables). For such abstract domains, an algorithm for $\widehat{\alpha}$ provides an algorithm for automatically synthesizing abstract transformers: given concrete transformer τ, encode the semantics of τ as a formula $\varphi_\tau \in \mathcal{L}$, and return $\widehat{\alpha}(\varphi_\tau)$ [31,36,34].

Let $\mathcal{G}_1 = \mathcal{C} \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \mathcal{A}_1$ and $\mathcal{G}_2 = \mathcal{C} \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \mathcal{A}_2$ be two Galois connections. We use $\mathcal{A}_1[V_1] \star \mathcal{A}_2[V_2]$ to denote the reduced product of the domains [8, §10.1], and $\langle A_1; A_2 \rangle$ to denote an element of $\mathcal{A}_1[V_1] \star \mathcal{A}_2[V_2]$.

4 Base Domains

All numeric-valued variables hold bit-vectors of bit-width w, and their values range over the set $\mathbb{Z}_{2^w} = \{0, \ldots, 2^w - 1\}$. All arithmetic is integer arithmetic modulo 2^w. Let $\mathtt{mm}$ denote a memory map, and $\mathtt{mm}[t]$ denote the w-bit value at memory address t. For instance, for Intel x86 machine code, $\mathtt{mm}[t]$ denotes (the little-endian interpretation of) the four bytes in memory pointed to by term t (cf. Ex. 1).

Bit-Vector Memory Domain ($\mathcal{M}$). Domain $\mathcal{M}$ can capture a limited class of invariants involving memory accesses. In particular, the domain is capable of capturing the constraint that the value of a variable v equals the value of the memory $\mathtt{mm}$ at address e: $v = \mathtt{mm}[e]$.

Definition 1. *Given variables P and memory map $\mathtt{mm}$, an element M of the **bit-vector memory domain** $\mathcal{M}[(\mathtt{mm}, P)]$ is either (i) $\bot$, or (ii) a set of constraints, where each constraint C_i is of the form $v_i = \mathtt{mm}[\Sigma_j a_{ij} v_j + b_i]$, $a_{ij}, b_i \in \mathbb{Z}_{2^w}$, $v_i, v_j \in P$. The concretization of M is*

$$\gamma(M) = \{(\mathtt{mm}, \overrightarrow{v}) \mid \textstyle\bigwedge_{C_i \in M} (\mathtt{mm}, \overrightarrow{v}) \models C_i\}.$$

□

The join and meet operation are defined as intersection and union of constraints, respectively. The join operation $\sqcup_\mathcal{M}$ is: $M_1 \sqcup_\mathcal{M} M_2 \overset{\text{def}}{=} \{C \mid C \in M_1 \text{ and } C \in M_2\}$. The meet operation $\sqcap_\mathcal{M}$ is: $M_1 \sqcap_\mathcal{M} M_2 \overset{\text{def}}{=} \{C \mid C \in M_1 \text{ or } C \in M_2\}$. The domain constructor for the bit-vector memory domain is defined as $\mathfrak{C}_\mathcal{M}(V_1, V_2) \overset{\text{def}}{=} \mathcal{M}[V_1 \uplus V_2]$.

Bit-Vector Equality Domain ($\mathcal{E}$). Domain $\mathcal{E}$ can capture relational affine equalities over bit-vectors [13].

Definition 2. *Given variables V, an element E of the **bit-vector equality domain** $\mathcal{E}[V]$ is either (i) $\bot$, or (ii) a set of affine equalities, where each equality C_i is of the form $\Sigma_j a_{ij} v_j + b_i = 0$, $a_{ij}, b_i \in \mathbb{Z}_{2^w}$, $v_j \in V$. The concretization of E is*

$$\gamma(E) = \{\overrightarrow{v} \mid \textstyle\bigwedge_{C_i \in E} \overrightarrow{v} \models C_i\}.$$

□

Elder et al. [13] introduced a normal form for representing elements of $\mathcal{E}[V]$, and used the normal form to give polynomial-time algorithms for the operations of join ($\sqcup_{\mathcal{E}}$), meet ($\sqcap_{\mathcal{E}}$), approximation order ($\sqsubseteq_{\mathcal{E}}$), and projection. The domain constructor for the bit-vector equality domain is defined as $\mathfrak{C}_{\mathcal{E}}(V_1, V_2) \overset{\text{def}}{=} \mathcal{E}[V_1 \uplus V_2]$.

Bit-Vector Interval Domain ($\mathcal{I}$). Domain $\mathcal{I}$ can capture non-relational bit-vector interval constraints.

Definition 3. *Given variables V, an element I of the* ***bit-vector interval domain*** *$\mathcal{I}[V]$ is either (i) $\bot$, or (ii) a set of interval constraints, where each constraint C_i is of the form $l_i \leq v_i \leq u_i, l_i, u_i \in \mathbb{Z}_{2^w}, v_i \in V$. The concretization of I is*

$$\gamma(I) = \{\overrightarrow{v} \mid \textstyle\bigwedge_{C_i \in I} \overrightarrow{v} \models C_i\}.$$

$\square$

The domain constructor for the bit-vector interval domain is defined as $\mathfrak{C}_{\mathcal{I}}(V_1, V_2) \overset{\text{def}}{=} \mathcal{I}[V_1]$.

5 The View-Product Combinator

In this section, we define the view-product combinator $\mathcal{V}$, and construct three domains using different applications of $\mathcal{V}$. $\mathcal{V}$ constructs a reduced-product of domains $\mathcal{A}_1$ and $\mathcal{A}_2$ so that a set of shared view-variables communicates information between the domains. $\mathcal{V}$ makes use of two principles:

Principle 1: View-variables are constrained by view-constraints. Consider abstract domain $\mathcal{A}_1[V_1]$, and let $V_1 \cap V_2 = \emptyset$. The variables in V_2 are the *view-variables.* $\mathcal{V}$ will use the domain $\mathcal{A}_1[V_1] \oplus V_2 = \mathcal{A}_1[V_1 \uplus V_2]$ (cf. §3). A *view-constraint* C for V_2 is an element of $\mathcal{A}[V_1 \uplus V_2]$ that serves to constrain the variable set V_2.

Principle 2: A view-constraint does not constrain the values of variables in V_1. Given abstract domain $\mathcal{A}[V_1 \uplus V_2]$, an *acceptable view-constraint* C for V_2 is an abstract value $C \in \mathcal{A}[V_1 \uplus V_2]$ such that $C \downarrow_{V_1} = \top$.

We are now in a position to describe how $\mathcal{V}$ works.

Arguments. $\mathcal{V}$ takes four arguments:
- $\mathcal{A}_1[V_1]$, an abstract domain defined over vocabulary V_1,
- V_2, a vocabulary such that $V_1 \cap V_2 = \emptyset$,
- $C \in \mathcal{A}_1[V_1 \uplus V_2]$, a view-constraint for V_2, and
- $\mathfrak{C}_{\mathcal{A}2}$, an abstract-domain constructor for abstract domain $\mathcal{A}_2$,

Enforcement of view-constraint C. The view-constraint C constrains the variables in V_2. Therefore, $\mathcal{V}$ should only consider those elements $\mathcal{A}[V_1 \uplus V_2]$ that satisfy C; that is, only elements $A \in \mathcal{A}[V_1 \uplus V_2]$ such that $A \sqsubseteq_{\mathcal{A}[V_1 \uplus V_2]} C$ holds. Put another way, the view-constraint C can be seen as an *integrity constraint.* The next definition formalizes this notion (for a general integrity constraint $D \in \mathcal{A}[V]$):

Definition 4. *Given an abstract domain $\mathcal{A}[V]$ and an element $D \in \mathcal{A}[V]$, the **abstract domain** $\mathcal{A}[V]$ **modulo** D, denoted by $\mathcal{A}[V]\,|_D$, is an abstract domain $\mathcal{A}'[V]$ that contains exactly the elements $D \sqcap A$, $A \in \mathcal{A}$; that is,*

$$\mathcal{A}[V]\,|_D \stackrel{\text{def}}{=} \{D \sqcap A \mid D \in \mathcal{A}[V]\}$$

□

Using the notation from Defn. 4, the domain that only contains elements that satisfy view-constraint C is $\mathcal{A}_1[V_1 \uplus V_2]\,|_C$.

Reduced product. Let $\mathcal{A}'_1[V_1 \uplus V_2]$ be $(\mathcal{A}_1[V_1] \oplus V_2)\,|_C$, where C is the view-constraint for view-variables V_2. All that is left is to perform a reduced product of $\mathcal{A}'_1$ and $\mathcal{A}_2$. However, for $\mathcal{A}'_1$ and $\mathcal{A}_2$ to be able to exchange information, the vocabulary of $\mathcal{A}_2$ should include at least the view-variables V_2. This condition is satisfied by the domain $\mathfrak{C}_{\mathcal{A}2}(V_2, V_1)$ constructed using the abstract-domain constructor for $\mathcal{A}_2$ supplied as the fourth argument to $\mathcal{V}$ (cf. §3).

Summing up, the reduced product performed by $\mathcal{V}$ creates $\mathcal{A}_3[V_1 \uplus V_2] := \mathcal{A}'_1[V_1 \uplus V_2] \star \mathfrak{C}_{\mathcal{A}2}(V_2, V_1)$.

Definition 5 (View-Product Combinator ($\mathcal{V}$)). *Given*
- *$\mathcal{A}_1[V_1]$, an abstract domain defined over vocabulary V_1,*
- *V_2, a vocabulary such that $V_1 \cap V_2 = \emptyset$,*
- *$C \in \mathcal{A}_1[V_1 \uplus V_2]$, a view-constraint for V_2, and*
- *$\mathfrak{C}_{\mathcal{A}2}$, an abstract-domain constructor for abstract domain $\mathcal{A}_2$,*

*the **view-product combinator** $\mathcal{V}[\mathcal{A}_1[V_1], V_2, C, \mathfrak{C}_{\mathcal{A}2}]$ constructs a domain $\mathcal{A}_3[V_1 \uplus V_2]$ such that*

$$\mathcal{A}_3[V_1 \uplus V_2] \stackrel{\text{def}}{=} (\mathcal{A}_1[V_1] \oplus V_2)\,|_C \star \mathfrak{C}_{\mathcal{A}2}(V_2, V_1)$$

□

Instantiations. We now describe three applications of $\mathcal{V}$ that use the bit-vector domains defined in §4.

$\mathcal{V}$ applied to the bit-vector memory domain $\mathcal{M}$ and the bit-vector equality domain $\mathcal{E}$ constructs the *Bit-Vector Memory-Equality Domain $\mathcal{BVME}$*, a domain of bit-vector affine-equalities over variables and memory-values.

Definition 6 (Bit-Vector Memory-Equality Domain ($\mathcal{BVME}$)).

$$\mathcal{BVME}[(\mathtt{mm}, P \uplus U)] \stackrel{\text{def}}{=} \mathcal{V}[\mathcal{M}[(\mathtt{mm}, P)], U, C_m, \mathfrak{C}_{\mathcal{E}}],$$

where
- *$\mathcal{M}[(\mathtt{mm}, P)]$, the bit-vector memory domain over memory map* mm *and variables P (cf. Defn. 1),*
- *U, a vocabulary of variables such that $U \cap P = \emptyset$,*
- *$C_m \in \mathcal{M}[(\mathtt{mm}, P \uplus U)]$, a view-constraint for U, and*
- *$\mathfrak{C}_{\mathcal{E}}(V_1, V_2) \stackrel{\text{def}}{=} \mathcal{E}[V_1 \uplus V_2]$.*

□

$\mathcal{V}$ applied to the bit-vector equality domain $\mathcal{E}$ and the bit-vector interval domain $\mathcal{I}$ constructs the *Bit-Vector Inequality Domain* $\mathcal{BVI}$, a domain of bit-vector affine-equalities over variables.

Definition 7 (The Bit-Vector Inequality Domain ($\mathcal{BVI}$)).

$$\mathcal{BVI}[P \uplus S] \stackrel{\text{def}}{=} \mathcal{V}[\mathcal{E}[P], S, C_s, \mathfrak{C}_{\mathcal{I}}],$$

where

- $\mathcal{E}[P]$*, the bit-vector memory domain over variables* P *(cf. Defn. 2),*
- S*, a vocabulary of variables such that* $S \cap P = \emptyset$*,*
- $C_s \in \mathcal{E}[P \uplus S)]$*, a view-constraint for* S*, and*
- $\mathfrak{C}_{\mathcal{I}}(V_1, V_2) \stackrel{\text{def}}{=} \mathcal{I}[V_1]$.

□

The combinator $\mathcal{V}$ applied to the bit-vector memory-equality domain $\mathcal{BVME}$, and a bit-vector interval domain $\mathcal{I}$ constructs the *Bit-Vector Memory-Inequality Domain* $\mathcal{BVMI}$, a domain of relational bit-vector affine-inequalities over variables and memory.

Definition 8 (Bit-Vector Memory-Inequality Domain ($\mathcal{BVMI}$)).

$$\mathcal{BVMI}[(\mathtt{mm}, P \uplus U \uplus S)] \stackrel{\text{def}}{=} \mathcal{V}[\mathcal{BVME}[(\mathtt{mm}, P \uplus U)], S, C_s, \mathfrak{C}_{\mathcal{I}}]$$

where

- $\mathcal{BVME}[(\mathtt{mm}, P \uplus U)]$*, the bit-vector memory-equality domain over memory map* $\mathtt{mm}$ *and variables* $P \uplus S$ *(cf. Defn. 6),*
- S*, a vocabulary of variables such that* $S \cap (P \uplus U) = \emptyset$*,*
- $C_s \in \mathcal{BVME}[(\mathtt{mm}, P \uplus U)]$*, a view-constraint for* S*, and*
- $\mathfrak{C}_{\mathcal{I}}(V_1, V_2) \stackrel{\text{def}}{=} \mathcal{I}[V_1]$.

□

6 Synthesizing Abstract Operations for Reduced-Product Domains

In this section, we first discuss a method for automatically synthesizing abstract operations for a general reduced-product domain $\mathcal{A}_3[V_1 \uplus V_2] := \mathcal{A}_1[V_1] \star \mathcal{A}_2[V_2]$ using the abstract operations of $\mathcal{A}_1$ and $\mathcal{A}_2$, which themselves may be automatically synthesized. We then discuss some pragmatic choices that we made for the reduced-product domains that the view-product combinator $\mathcal{V}$ creates.

Synthesizing Abstract Transformers. If $\langle A_1; A_2\rangle \in \mathcal{A}_1 \star \mathcal{A}_2 = \mathcal{A}_3$, then $\widehat{\gamma}_3(\langle A_1; A_2\rangle) = \widehat{\gamma}_1(A_1) \wedge \widehat{\gamma}_2(A_2)$, and $\widehat{\alpha}_3(\varphi) = \langle \widehat{\alpha}_1(\varphi); \widehat{\alpha}_2(\varphi)\rangle$. As mentioned in §3, for an abstract domain whose elements are abstractions of transition relations, an algorithm for $\widehat{\alpha}$ provides an algorithm for automatically synthesizing abstract transformers: given concrete transformer τ, encode the semantics of τ

as a formula $\varphi_\tau \in \mathcal{L}$, and return $\widehat{\alpha}(\varphi_\tau)$. Moreover, as long as one of $\mathcal{A}_1$ or $\mathcal{A}_2$ is an abstract transition-relation domain, then so is $\mathcal{A}_3 := \mathcal{A}_1 \star \mathcal{A}_2$. Thus, if we are given $\widehat{\gamma}_1$, $\widehat{\alpha}_2$, $\widehat{\gamma}_2$, and $\widehat{\alpha}_2$, we obtain $\widehat{\gamma}_3$ and $\widehat{\alpha}_3$, and $\widehat{\alpha}_3$ gives us a way to synthesize abstract transformers for $\mathcal{A}_3$.

Semantic Reduction. The semantic reduction of $\langle A_1; A_2\rangle \in \mathcal{A}_1 \star \mathcal{A}_2 = \mathcal{A}_3$ can be computed as $\widehat{\alpha}_3(\psi)$, where ψ is $\widehat{\gamma}_1(A_1) \wedge \widehat{\gamma}_2(A_2)$. Computing the semantic reduction in this way can be computationally expensive. Instead we assume there exists a *weak semantic-reduction* operator `Reduce`. Using this weak semantic-reduction operator we can define the other abstract operations for the reduced-product domain. Furthermore, because it can be expensive to determine whether $\langle A_1; A_2\rangle \sqsubseteq \langle A_1'; A_2'\rangle$ holds, we define a weaker approximation order $\widetilde{\sqsubseteq}$:

Definition 9. *The* ***weak approximation order*** $\widetilde{\sqsubseteq}$ *is defined as follows:* $\langle A_1'; A_2'\rangle \,\widetilde{\sqsubseteq}\, \langle A_1''; A_2''\rangle$ *if and only if* $A_1' \sqsubseteq_{\mathcal{A}_1} A_1''$ *and* $A_2' \sqsubseteq_{\mathcal{A}_2} A_2''$. □

It is easy to show that if $\langle A_1'; A_2'\rangle \,\widetilde{\sqsubseteq}\, \langle A_1''; A_2''\rangle$, then $\langle A_1'; A_2'\rangle \sqsubseteq \langle A_1''; A_2''\rangle$, though the converse may not always hold.

We define *quasi-join*, an approximation to the join operator for reduced-product domain, which is not guaranteed to return the least-upper bound, but is sound and simple.

Definition 10. *The* ***quasi-join*** *operator, denoted by* $\widetilde{\sqcup}$, *is defined as follows:* $\langle A_1'; A_2'\rangle \,\widetilde{\sqcup}\, \langle A_1''; A_2''\rangle \overset{\text{def}}{=} \texttt{Reduce}(\langle A_1' \sqcup_{\mathcal{A}_1} A_1''; A_2' \sqcup_{\mathcal{A}_2} A_2''\rangle)$. □

Theorem 1. **[Soundness of Quasi-Join]** *Let* $\langle A_1; A_2\rangle = \langle A_1'; A_2'\rangle \,\widetilde{\sqcup}\, \langle A_1''; A_2''\rangle$. *Then* $\gamma(\langle A_1; A_2\rangle) \supseteq \gamma(\langle A_1'; A_2'\rangle) \cup \gamma(\langle A_1''; A_2''\rangle)$. □

The *quasi-meet* operation is similar in flavor to quasi-join:

Definition 11. *The* ***quasi-meet*** *operator, denoted by* $\widetilde{\sqcap}$, *is defined as follows:* $\langle A_1'; A_2'\rangle \,\widetilde{\sqcap}\, \langle A_1''; A_2''\rangle \overset{\text{def}}{=} \texttt{Reduce}(\langle A_1' \sqcap_{\mathcal{A}_1} A_1''; A_2' \sqcap_{\mathcal{A}_2} A_2''\rangle)$. □

Theorem 2. **[Soundness of Quasi-Meet]** *Let* $\langle A_1; A_2\rangle = \langle A_1'; A_2'\rangle \,\widetilde{\sqcap}\, \langle A_1''; A_2''\rangle$. *Then* $\gamma(\langle A_1; A_2\rangle) \supseteq \gamma(\langle A_1'; A_2'\rangle) \cap \gamma(\langle A_1''; A_2''\rangle)$. □

We now define weak semantic-reduction operators for each of the pairs of domains used in our constructions. The algorithms for these specific domains have not been stated explicitly in the literature, and are stated here for completeness. (Previous work tackled the rational-arithmetic variants of these domains.)

Weak Reduce for $\mathcal{M} \star \mathcal{E}$. Alg. 1 computes the weak semantic-reduction for the bit-vector memory domain and bit-vector equality domain. Given $\langle M; E\rangle \in \mathcal{M}[(\texttt{mm}, V)] * \mathcal{E}[V]$, $\texttt{Reduce}(\langle M; E\rangle)$ infers further equalities among V. The key insight is to model the memory map `mm` as an uninterpreted function; that is, $t_1 = t_2$ implies $\texttt{mm}[t_1] = \texttt{mm}[t_2]$. We are effectively approximating the theory of arrays using the theory of uninterpreted functions with equality (EUF) [5], thereby ensuring that the reduction operation is efficient. As shown in lines (3) and (4), if $e_1 = e_2$ then the algorithm infers that $v_1 = v_2$. The following example illustrates the working of Alg. 1.

Algorithm 1: $\texttt{Reduce}_{\mathcal{M}\star\mathcal{E}}(\langle M; E\rangle)$

```
1 foreach constraint $v_1 = \texttt{mm}[e_1] \in M$ do
2   foreach constraint $v_2 = \texttt{mm}[e_2] \in M$ do
3     if $E \sqsubseteq_{\mathcal{E}} \{e_1 = e_2\}$ then
4       $E \leftarrow E \sqcap_{\mathcal{E}} \{v_1 = v_2\}$
5 return $\langle M; E\rangle$
```

Fig. 3: Algorithm for weak semantic-reduction for $\mathcal{M}[V] \star \mathcal{E}[V]$.

Example 5. Let V be $\{x, y, z, u_1, u_2\}$. Consider $\langle M; E\rangle := \langle u_1 = \texttt{mm}[x], u_2 = \texttt{mm}[y + 8]; x = y + z - 2, z = 10\rangle$. From E we can infer that $x = y + 8$. Thus, $\texttt{mm}[x] = \texttt{mm}[y + 8]$, and we can infer that $u_1 = u_2$. Thus, E can be updated to $E \sqcap_{\mathcal{E}} \{u_1 = u_2\}$.

No further reduction is possible; thus, $\texttt{Reduce}_{\mathcal{M}\star\mathcal{E}}(\langle M; E\rangle = \langle u_1 = \texttt{mm}[x], u_2 = \texttt{mm}[y + 8]; u_1 = u_2, x = y + z - 2, z = 10\rangle$. □

The following example shows a case when $\texttt{Reduce}_{\mathcal{M}\star\mathcal{E}}$ fails to find the most precise answer.

Example 6. Consider the situation when analyzing Intel x86 machine code. The bit-width w is 32. The memory map $\texttt{mm}$ is a map from 32-bit bit-vector to 8-bit bit-vectors, and the addressing mode is little-endian (cf. Ex. 1). Let V be $\{r, u_1, u_2, u_3\}$. Consider $\langle M; E\rangle := \langle u_1 = \texttt{mm}[r], u_2 = \texttt{mm}[r + 2], u_3 = \texttt{mm}[r + 4]; u_1 = 0, u_3 = 0; \top\rangle$. Because neither $r = r + 2$ nor $r = r + 4$ hold, Alg. 1 cannot infer any further equalities among u_1, u_2, and u_3. Thus, $\texttt{Reduce}_{\mathcal{M}\star\mathcal{E}}(\langle M; E\rangle) = \langle M; E\rangle$.

However, $\langle M; E\rangle$ is not the best reduction possible. Because $u_1 = 0$ and $u_3 = 0$, we can infer that $\texttt{mm}[r]$ and $\texttt{mm}[r + 4]$ are 0. Thus, the bytes at addresses r through $r + 7$ are all 0, because we assumed little-endian addressing. Consequently, $\texttt{mm}[r + 2] = 0$, and $u_2 = 0$. Thus, the best reduction for $\langle M; E\rangle$ is $\langle u_1 = \texttt{mm}[r], u_2 = \texttt{mm}[r + 2], u_3 = \texttt{mm}[r + 4]; u_1 = 0, u_2 = 0, u_3 = 0\rangle$. The reason the $\texttt{Reduce}_{\mathcal{M}\star\mathcal{E}}$ algorithm was unable to deduce this was because the algorithm treats the memory map as an uninterpreted function. Thus, though sound, Alg. 1 is not always able to deduce the best possible reduced value. □

Theorem 3. [Soundness of Alg. 1] *Let* $\langle M'; E'\rangle = \texttt{Reduce}_{\mathcal{M}\star\mathcal{E}}(\langle M; E\rangle)$. *Then* $\gamma(\langle M'; E'\rangle) = \gamma(\langle M; E\rangle)$, *and* $\langle M'; E'\rangle \,\widetilde{\sqsubseteq}\, \langle M; E\rangle)$. □

Weak Reduce for $\mathcal{E} \star \mathcal{I}$. Alg. 2 computes a weak semantic-reduction for the bit-vector equality domain and the bit-vector interval domain. Given $\langle E; I\rangle \in \mathcal{E}[P \uplus S] \star \mathcal{I}[S]$, $\texttt{Reduce}_{\mathcal{E}\star\mathcal{I}}(\langle E; I\rangle)$ infers tighter interval bounds on the variables in S. Reduction is performed by first projecting E onto S to determine the affine relations E' that hold among the variables in S (line (1)). These affine relations are used in lines (2)–(5) to infer tighter intervals for the variables in S. "$[\![expr]\!]$" denotes the evaluation of expression *expr* over interval domain $\mathcal{I}$ via interval arithmetic. New interval constraints are identified by evaluating expressions of

Algorithm 2: $\texttt{Reduce}_{\mathcal{E}\star\mathcal{I}}(\langle E; I\rangle)$

```
1 $E' \leftarrow E\downarrow_S$
2 **foreach** $s \in S$ **do**
3     **foreach constraint** $(s = c + \sum_{t\in S\wedge t\neq s} a_t t) \in E'$ **do**
4         $\iota \leftarrow [\![c + \sum_{t\in S\wedge t\neq s} a_t t]\!](I)$
5         $I \leftarrow I \sqcap_{\mathcal{I}} \top[s \in \iota]$
6 **return** $\langle E; I\rangle$
```

Fig. 4: Algorithm for weak semantic-reduction for $\mathcal{E}[P \uplus S] \star \mathcal{I}[S]$.

Program	Measures of Size				Performance (sec.)		Precision ($\mathcal{BVI} \sqsubset \mathcal{E}$)	
Name	Instrs	CFGs	BBs	Branches	$\mathcal{E}$	$\mathcal{BVI}$	Control-Point Invariants	Procedure Summaries
finger	532	18	298	48	185	639	24/48	3/18
subst	1093	16	609	74	303	1151	11/74	3/16
label	1167	16	573	103	236	986	24/103	2/16
chkdsk	1468	18	787	119	631	1675	12/119	3/18
convert	1927	38	1013	161	441	1744	101/161	0/38
route	1982	40	931	243	749	2497	78/243	2/39
comp	2377	35	1261	224	849	2740	22/224	0/35
logoff	2470	46	1145	306	861	3253	124/306	13/46

Table 1: Machine-code analysis using $\mathcal{BVI}$. Columns 6–9 show the times (in seconds) for the $\mathcal{E}$-based analysis, and for the $\mathcal{BVI}$-based analysis; and the degree of improvement in precision measured as the number of control points at which $\mathcal{BVI}$-based analysis gave more precise invariants compared to $\mathcal{E}$-based analysis, and the number of procedures for which $\mathcal{BVI}$-based analysis gave more precise summaries compared to $\mathcal{E}$-based analysis.

the form $c + \sum_{t\in S\wedge t\neq s} a_t t$ over $\mathcal{I}$ using the current interval value I (line (4)). These constraints are then incorporated into I via meet (line (5)).

Example 7. Let $P := \{x, y\}$, and $S := \{s_1, s_2\}$, where the bit-width of the bit-vector is 4. Consider $\langle E; I\rangle := \langle s_1 = 2x + 2y, s_2 = x + y; s_1 \in [4, 9], s_2 \in [3, 5]\rangle$. By projecting E onto the variables $S := \{s_1, s_2\}$, we obtain $E' = \{s_1 = 2s_2\}$ on line (1). On line (4), we have $\iota = [\![2s_2]\!](I) = [6, 10]$. Using this equation, I is updated on line (5); that is, $I = \{s_1 \in [4, 9], s_2 \in [3, 5]\} \sqcap \{s_1 \in [6, 10], s_2 \in \top\} = \{s_1 \in [6, 9], s_2 \in [3, 5]\}$. No further reduction is possible.

Thus, $\texttt{Reduce}_{\mathcal{E}\star\mathcal{I}}(\langle E; I\rangle) = \langle s_1 = 2x + 2y, s_2 = x + y; s_1 \in [6, 9], s_2 \in [3, 5]\rangle$. Note that Alg. 2 is not guaranteed to deduce the best possible interval constraints. The best possible reduction of $\langle E; I\rangle$ is $\langle s_1 = 2x + 2y, s_2 = x + y; s_1 \in [6, 8], s_2 \in [3, 4]\rangle$. □

Theorem 4. [Soundness of Alg. 2] *Let $\langle E'; I'\rangle = \texttt{Reduce}_{\mathcal{E}\star\mathcal{I}}(\langle E; I\rangle)$. Then $\gamma(\langle E'; I'\rangle) = \gamma(\langle E; I\rangle)$, and $\langle E'; I'\rangle \,\widetilde{\sqsubseteq}\, \langle E; I\rangle$.* □

7 Experimental Evaluation

In this section, we compare the performance and precision of the bit-vector equality domain $\mathcal{E}$ with that of the bit-vector inequality domain $\mathcal{BVI}$. The abstract transformers for the $\mathcal{BVI}$ domain were synthesized using the approach given in Section 6. The weak semantic-reduction operator described in Alg. 2 was used in the implementation of the compose operation needed for interprocedural analysis.

Experimental Setup. We analyzed a corpus of Windows utilities using the WALi [19] system for weighted pushdown systems (WPDSs). Tab. 1 lists several size parameters of the examples (number of instructions, procedures, basic blocks, and branches).[3] The weight on each WPDS rule encodes the abstract transformer for a basic block B of the program, including a jump or branch to a successor block. A formula φ_B is created that captures the concrete semantics of B, and then the weight for B is obtained by performing $\widehat{\alpha}(\varphi_B)$. We used EWPDS merge functions [22] to preserve caller-save and callee-save registers across call sites. The post* query used the FWPDS algorithm [21].

View-selection Heuristic. Given the set of machine registers P, the view-constraints C_s were computed as $C_s := \bigcup_{r_i \in P} \{s_{i1} = r_i, s_{i2} = r_i + 2^{31}\}$. In particular, the view-variable s_{i2} allows us to keep track of whether r_i, when treated as a signed value, is less-than or equal 0.

Performance. Columns 6 and 7 of Tab. 1 list the time taken, in seconds, for $\mathcal{E}$-based analysis, and the $\mathcal{BVI}$-based analysis. On average (geometric mean), $\mathcal{BVI}$-based analysis is about 3.5 times slower than $\mathcal{E}$-based analysis.

Precision. We compare the procedure summaries, and the invariants for each control point—i.e., the point just before a branch instruction. Column 8 lists the number of control points at which $\mathcal{BVI}$-based analysis gave more precise invariants compared to $\mathcal{E}$-based analysis. $\mathcal{BVI}$-based analysis gives more precise invariants at up to 63% of control points, and, on average, $\mathcal{BVI}$-based analysis improves precision for 29% of control points. Column 9 lists the number of procedures for which $\mathcal{BVI}$-based analysis gave more precise summaries compared to $\mathcal{E}$-based analysis. $\mathcal{BVI}$-based analysis gives more precise summaries for up to 17% of procedures, and, on average $\mathcal{BVI}$-based analysis gave better summaries for 9.3% of procedures.

8 Related Work

To construct bit-vector *inequality* domains, we made use of $\mathcal{E}$, a relational abstract domains for affine *equality* constraints over modular arithmetic, originally defined by King and Søndergaard [20] and later improved by Elder et al. [13]. The main reason for using $\mathcal{E}$, as opposed to other abstract domains that can capture affine equality constraints over modular arithmetic, such as the domains of Müller-Olm and Seidl [27] and Granger [17, 16], is that we know how to perform $\widehat{\alpha}$ for $\mathcal{E}$ [13], but do not know how to perform $\widehat{\alpha}$ for the other domains cited.

Other work on identifying bit-vector-inequality invariants includes Brauer and King [3, 4] and Masdupuy [25]. Masdupuy proposed a relational abstract

[3] Due to the high cost of the WPDS construction, all analyses excluded the code for libraries. Because register `eax` holds the return value from a call, library functions were modeled approximately (albeit unsoundly, in general) by "`havoc(eax)`".

domain of interval congruences on rationals. One limitation of his machinery is that the domain represents diagonal grids of parallelepipeds, where the dimension of each parallelepiped equals the number of variables tracked (say n). In our work, we can have any number of view-variables, which means that the point-spaces represented can be constrained by more than n constraints.

Brauer and King employ bit-blasting to synthesize abstract transformers for the interval and octagon [26] domains. One of their papers uses universal-quantifier elimination on Boolean formulas [3]; the other avoids quantifier elimination [4]. Compared with their work, we avoid the use of bit-blasting and work directly with representations of sets of w-bit bit-vectors. The greatly reduced number of variables that comes from working at word level opens up the possibility of applying our methods to much larger problems; as discussed in §7, we were able to apply our methods to interprocedural program analysis. The equality domain $\mathcal{E}$ that we work with can capture relations on an arbitrary number of variables, and thus so can the domains that we construct using $\mathcal{V}$. Compared with octagons, which are limited to two variables and coefficients of ± 1, the advantage is that our domains can express more interesting invariants and procedure summaries. In particular, Octagon-based summaries would be limited to one pre-state variable and one post-state variable.

The view-product combinator $\mathcal{V}$ is a compositional generalization of the construction used in SubPoly. SubPoly is constructed as a reduced product of an affine-equality domain $\mathcal{K}$ over rationals [18], and an interval domain $\mathcal{J}$ over rationals [7] with slack view-variables S to communicate between the two domains. SubPoly can be constructed by applying $\mathcal{V}$ to $\mathcal{K}$ and $\mathcal{J}$, as follows:

$$\text{SubPoly}[P \uplus S] \stackrel{\text{def}}{=} \mathcal{V}[\mathcal{K}[P], S, C_s, \mathfrak{J}], \tag{1}$$

where C_s is the view-constraint for S and $\mathfrak{J}(V_1, V_2) \stackrel{\text{def}}{=} \mathcal{J}[V_1]$.

When an analysis system works with two or more reasoning techniques, there is often an opportunity to share information to improve the precision of both. The principle is found in the classic papers of Cousot and Cousot [8] and Nelson and Oppen [28]. In practice, there are a range of choices as to what might be shared, and our work represents one point in that design space. The algorithms for weak semantic-reduction (Algs. 1 and 2) adapt techniques for theory combination that have been used in Satisfiability Modulo Theory (SMT) solvers [12, 14].

The work of Chang and Leino [6] is similar in spirit to ours. They developed a technique for extending the properties representable by a given abstract domain from schemas over variables to schemas over terms. To orchestrate the communication of information between domains, they designed the congruence-closure abstract domain, which introduces variables to stand for subexpressions that are alien to a base domain; to the base domain, these expressions are just variables. Their scheme for propagating information between domains is mediated by the e-graph of the congruence-closure domain. In contrast, our method can make use of past work on synthesizing best abstract operations [31, 34] to propagate information between domains. As discussed in §6, we also employ less precise, more pragmatic procedures that use information in one domain to iteratively refine information in another domain. Cousot et al. [9] have recently studied the iterated pairwise exchange of observations between components as a way to compute an overapproximation of a reduced product.

9 Conclusion

The key contribution of the paper is to show that once you solve a lot of fundamental problems (bit-vector equality domain, bit-vector interval domain, bit-vector memory domain, automatic synthesis of abstract transformers), you can use the view-product combinator to solve the complex problem of designing a bit-vector domain that (i) models memory and (ii) can capture affine inequalities.

Preliminary experimental results based on the $\mathcal{BVI}$ domain illustrate the practical benefits of our approach. In the future, we intend to evaluate the $\mathcal{BVME}$ and $\mathcal{BVMI}$ domains.

References

1. G. Balakrishnan and T. Reps. WYSINWYX: What You See Is Not What You eXecute. *TOPLAS*, 2010.
2. J. Bloch. Extra, extra - read all about it: Nearly all binary searches and mergesorts are broken. "googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html".
3. J. Brauer and A. King. Automatic abstraction for intervals using Boolean formulae. In *SAS*, 2010.
4. J. Brauer and A. King. Transfer function synthesis without quantifier elimination. In *ESOP*, 2011.
5. J. Burch and D. Dill. Automatic verification of pipelined microprocessor control. In *CAV*, 1994.
6. B.-Y. Chang and K. Leino. Abstract interpretation with alien expressions and heap structures. In *VMCAI*, 2005.
7. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. 2nd. Int. Symp on Programming*, Paris, Apr. 1976.
8. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, 1979.
9. P. Cousot, R. Cousot, and L. Mauborgne. The reduced product of abstract domains and the combination of decision procedures. In *FOSSACS*, 2011.
10. P. Cousot and N. Halbwachs. Automatic discovery of linear constraints among variables of a program. In *POPL*, 1978.
11. dspace targetlink. "www.dspaceinc.com/en/inc/home/products/sw/pcgs/targetli.cfm".
12. B. Dutertre and L. de Moura. A fast linear-arithmetic solver for dpll(t). In *CAV*, 2006.
13. M. Elder, J. Lim, T. Sharma, T. Andersen, and T. Reps. Abstract domains of affine relations. In *SAS*, 2011.
14. V. Ganesh and D. Dill. A decision procesure for bit-vectors and arrays. In *CAV*, 2007.
15. H. L. Garner. Theory of computer addition and overflow. *IEEE Trans. on Computers*, C-27(4), Apr. 1978.
16. P. Granger. Static analysis of arithmetic congruences. *Int. J. of Comp. Math.*, 1989.

17. P. Granger. *Analyses Semantiques de Congruence*. PhD thesis, Ecole Polytechnique, 1991.
18. M. Karr. Affine relationship among variables of a program. *Acta Inf.*, 6, 1976.
19. N. Kidd, A. Lal, and T. Reps. WALi: The Weighted Automaton Library, 2007. www.cs.wisc.edu/wpis/wpds/download.php.
20. A. King and H. Søndergaard. Automatic abstraction for congruences. In *VMCAI*, 2010.
21. A. Lal and T. Reps. Improving pushdown system model checking. In *CAV*, 2006.
22. A. Lal, T. Reps, and G. Balakrishnan. Extended weighted pushdown systems. In *CAV*, 2005.
23. V. Laviron and F. Logozzo. Subpolyhedra: A (more) scalable approach to infer linear inequalities. In *VMCAI*, 2009.
24. J. Lim and T. Reps. A system for generating static analyzers for machine instructions. In *CC*, 2008.
25. F. Masdupuy. Array abstractions using semantic analysis of trapezoid congruences. In *ICS*, 1992.
26. A. Miné. The octagon abstract domain. In *WCRE*, 2001.
27. M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. In *ESOP*, 2005.
28. G. Nelson and D. Oppen. Simplification by cooperating decision procedures. *TOPLAS*, 1(2), 1979.
29. T. Reps, G. Balakrishnan, and J. Lim. Intermediate-representation recovery from low-level code. In *Part. Eval. and Semantics-Based Prog. Manip.*, 2006.
30. T. Reps, J. Lim, A. Thakur, G. Balakrishnan, and A. Lal. There's plenty of room at the bottom: Analyzing and verifying machine code. In *CAV*, 2010.
31. T. Reps, M. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *VMCAI*, 2004.
32. R. Sen and Y. Srikant. Executable analysis using abstract interpretation with circular linear progressions. In *MEMOCODE*, 2007.
33. A. Simon and A. King. Taming the wrapping of integer arithmetic. In *SAS*, 2007.
34. A. Thakur, M. Elder, and T. Reps. Bilateral algorithms for symbolic abstraction. In *SAS*, 2012.
35. A. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. Reps. Directed proof generation for machine code. In *CAV*, 2010.
36. A. Thakur and T. Reps. A method for symbolic computation of abstract operations. In *CAV*, 2012.
37. H. Warren, Jr. *Hacker's Delight*. Addison-Wesley, 2003.