Program Analysis via Efficient Symbolic Abstraction

PEISEN YAO*, The Hong Kong University of Science and Technology, China
QINGKAI SHI, Ant Group, China
HEQING HUANG, The Hong Kong University of Science and Technology, China
CHARLES ZHANG, The Hong Kong University of Science and Technology, China

This paper concerns the scalability challenges of symbolic abstraction: given a formula φ in a logic $\mathcal{L}$ and an abstract domain $\mathcal{A}$, find a most precise element in the abstract domain that over-approximates the meaning of φ. Symbolic abstraction is an important point in the space of abstract interpretation, as it allows for automatically synthesizing the best abstract transformers. However, current techniques for symbolic abstraction can have difficulty delivering on its practical strengths, due to performance issues.

In this work, we introduce two algorithms for the symbolic abstraction of quantifier-free bit-vector formulas, which apply to the bit-vector interval domain and a certain kind of polyhedral domain, respectively. We implement and evaluate the proposed techniques on two machine code analysis clients, namely static memory corruption analysis and constrained random fuzzing. Using a suite of 57,933 queries from the clients, we compare our approach against a diverse group of state-of-the-art algorithms. The experiments show that our algorithms achieve a substantial speedup over existing techniques and illustrate significant precision advantages for the clients. Our work presents strong evidence that symbolic abstraction of numeric domains can be efficient and practical for large and realistic programs.

CCS Concepts: • **Theory of computation** → **Automated reasoning**; **Program analysis**.

Additional Key Words and Phrases: Abstract interpretation, symbolic abstraction, optimization, interval domain, polyhedral domain

ACM Reference Format:
Peisen Yao, Qingkai Shi, Heqing Huang, and Charles Zhang. 2021. Program Analysis via Efficient Symbolic Abstraction. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 118 (October 2021), 32 pages. https://doi.org/10.1145/3485495

1 INTRODUCTION

Abstract interpretation is a general theory for constructing sound static analysis by approximation [Cousot and Cousot 1979]. At its heart stands the concept of *abstract domain*, a mathematical representation of the program semantics. For instance, numerical domains such as octagon [Miné 2006] and polyhedron [Cousot and Halbwachs 1978] capture the numerical properties of program variables. Such information is useful for proving the absence of buffer overflow, division by zero, and many other properties [Blanchet et al. 2003; Singh et al. 2017a, 2015].

*Also with Ant Group.

Authors' addresses: Peisen Yao, The Hong Kong University of Science and Technology, China, pyao@cse.ust.hk; Qingkai Shi, Ant Group, China, qingkai.sqk@antgroup.com; Heqing Huang, The Hong Kong University of Science and Technology, China, hhuangaz@cse.ust.hk; Charles Zhang, The Hong Kong University of Science and Technology, China, charlesz@cse.ust.hk.

2475-1421/2021/10-ART118
https://doi.org/10.1145/3485495

1.1 Symbolic Abstraction

Given an abstract domain, the program analysis designers must provide abstract transformers that over-approximate the concrete semantics of various program statements, such as assignments and conditionals. A fundamental problem in abstract interpretation is to construct the *best*, i.e., the most precise abstract transformer [Cousot and Cousot 1979]. In their seminal work, Reps et al. [2004] introduce the problem of *symbolic abstraction*, which uses decision procedures for automatic construction of best transformers. Specifically, given a formula $\varphi \in \mathcal{L}$ encoding the concrete semantics, and an abstract domain $\mathcal{A}$, symbolic abstraction computes a most precise element in $\mathcal{A}$ that over-approximates the meaning of φ. To date, symbolic abstraction has found many applications, such as shape analysis [Reps et al. 2004; Yorsh et al. 2004], program verification [Jiang et al. 2017; Li et al. 2014], control flow recovery [Barrett and King 2010], and compiler optimization [Ritter 2015].

Conventionally, the abstract transformer for a block of code is obtained by composing the block's individual statements' abstract transformers. In comparison, the salient merit of symbolic abstraction is allowing for encoding the block as a formula φ, and analyzing the block as a whole to obtain the (best) abstract transformer.

Our target application is program analysis for low-level instructions such as assembly and x86, where the operators are built out of successions of small elementary instructions. On the one hand, analyzing low-level code allows reasoning about the actual behaviors of an executable program more faithfully, because the semantics of elementary low-level instructions are usually fairly well-defined [Dasgupta et al. 2019]. In contrast, the compiling processes of higher-level languages may leave significant leeway (known as the "What You See Is Not What You eXecute" phenomenon [Gopan and Reps 2007]). On the other hand, however, analyzing the low-level instructions is challenging and tedious, due to the reduced size of the code window used for transfer functions [Logozzo and Fähndrich 2008]. Therefore, it is important to be able to analyze program blocks as a whole, not as the composition of a succession of independent instructions [Barrett and King 2010; Brauer and King 2010]. Symbolic abstraction presents a way to tame the complexity by offering two key benefits:

- ***Precision***. It is well known that abstract interpretation is not compositional, meaning that the composition of the best abstract transformers of individual statements in a sequence may not result in the best abstract transformer for the whole sequence [Cousot and Cousot 1979]. To see how symbolic abstraction can yield better results than creating abstract transformers by composition, consider performing interval analysis on the code snippet: $x \in [0, 1]; y = x; z = x - y$. The interval of z, obtained from those for x and y by applying the rules of conventional interval arithmetics[1], is $z \in [-1, 1]$. However, the optimal interval, i.e., best abstraction is $z \in [0, 0]$, which can be computed via symbolic abstraction.
- ***Automation***. Conventionally, the static analysis developers need to design, implement, and tune abstract transformers for various program instructions, which can be tedious and error-pone. For example, a standard implementation of the polyhedral domain contains more than 40 operators [Singh et al. 2017b]. In contrast, symbolic abstraction allows for synthesizing a (correct and precise) abstract transformer for a block of code, instead of independently designing and composing abstract transformers for different instructions.

Despite the promise, however, there is a performance gap for current symbolic abstraction technology to be practical for real-world and large-scale programs. In theory, symbolic abstraction is computationally expensive. For instance, the best transformers for assignments in weakly relational domains such as octagon have the same worst-case exponential complexity as the polyhedral

[1]For instance, the statement $z = x + y$ is abstracted as $z_{max} = x_{max} + y_{max}$ and $z_{min} = x_{min} + y_{min}$.

domain [Singh et al. 2017c]. In practice, symbolic abstraction faces the scalability issue that limits its adoption. For example, it was reported that a state-of-the-art algorithm that symbolically computes best transformers for the affine-relation domain (ARA) can be more than 50× slower than a conventional method for computing sound, but not necessarily the best ARA transformers [Thakur and Reps 2012].

Recent advances in Optimization Modulo Theory (OMT) solving [Bjørner et al. 2015; Li et al. 2014; Nadel and Ryvchin 2016; Sebastiani and Trentin 2015a] provide new insights into the symbolic abstraction of template linear domains [Jiang et al. 2017; Li et al. 2014]. However, OMT is a young technology with large margins for improvement [Sebastiani and Trentin 2015a], and we observe that many real-world symbolic abstraction problems pose challenges to state-of-the-art OMT solvers. Besides, existing OMT-based solutions for symbolic abstraction are not directly applicable to the polyhedral domain (conjunctions of linear inequalities), because both the number of inequalities and the coefficients in each inequality are unknown prior.

1.2 Our Work

This paper aims to speed up the symbolic interval and polyhedral abstractions of quantifier-free bit-vector formulas. Our key technical insight is two-fold. First, the variables in programs are often correlated. Second, the interval and polyhedral domains in the bit-vector arithmetic are bounded. Taken together, they allow us to reduce the search space of symbolic abstraction. The main challenge, however, is how to effectively leverage the correlations and boundedness for reducing redundant computations.

We first present an analysis for computing the best interval abstraction. Our approach first conducts a static analysis to infer a sound abstraction, followed by an SMT-based refinement that iteratively finds the optimal intervals. Crucially, the second step refines all the variables synergistically, reusing information between them to speed up the analysis. Using our interval analysis, we then introduce a symbolic polyhedral analysis that interleaves the computations of intervals and polyhedrons. Our analysis builds on a novel integral polyhedral domain for bit-vector arithmetic (detailed in § 2.3). The key idea of the analysis is to utilize the interval abstractions to approach more "extremal" points near to the enclosing convex shape, which is a bounded integral polyhedron. Our algorithm opens up a new connection between OMT and symbolic abstraction, which allows for bringing OMT solving techniques to symbolic polyhedral abstraction.

We have implemented the proposed techniques as a tool called Taichi, and applied it in two applications of machine code analysis, namely static memory corruption analysis and constrained random fuzzing. Using 57,933 symbolic abstraction queries from the clients, we compare our techniques against two classes of algorithms from OMT solving and symbolic abstraction literature, respectively. The first class computes the best interval abstraction via OMT solving. We evaluate three OMT solvers that are based on MaxSAT solving [Nadel and Ryvchin 2016; Narodytska and Bacchus 2014] and that reduce to quantified formulas (e.g., [Kong et al. 2018]). The second class computes the best polyhedral abstraction, including algorithms due to Reps et al. [2004] (RSY), Thakur and Reps [2012] (TR), and Thakur et al. [2012] (TER).

This paper makes the following contributions to the symbolic abstraction of numeric domains:

- We present an efficient algorithm for symbolic interval abstraction. Our algorithm is on average 2.1× to 17.6× faster than several state-of-the-art OMT solvers.
- We present an efficient algorithm for the problem of symbolic polyhedral abstraction (as defined in § 2.3). When evaluated on non-overflowing formulas, our analysis is on average 3.6×, 2.6×, and 2.4× faster than RSY, TR, TER, respectively, and it solves more queries than the other three algorithms.

- We implement the proposed algorithms and demonstrate their usability for static memory corruption analysis and constrained random fuzzing.

2 BACKGROUND AND PROBLEM FORMULATION

This section introduces the background knowledge (symbolic abstraction and optimization modulo theories) and defines the problems we address in the paper.

2.1 Symbolic Abstraction

Abstract Interpretation. Let $(C, \leq_C)$ and $(A, \leq_A)$ be two complete lattices, a paired abstraction function $\alpha : C \to A$ and concretization function $\gamma : A \to C$ forms a *Galois connection* between C and A if for any $c \in C$ and $a \in A$, we have $\alpha(c) \leq_A a \Leftrightarrow c \leq_C \gamma(a)$. We call $\mathcal{A} = \langle A, \leq_A, \sqcup, \alpha, \gamma \rangle$ an abstract domain with the join operator $\sqcup$ and the partial order relation $\leq_A$. Given a concrete transfer function $f : C \to C$, we say an abstract function $f^\# : A \to A$ is a *sound abstraction* of f if $\alpha(f(c)) \leq_A f^\#(\alpha(c))$ for any $c \in C$. We say an abstract function f^α is the *best abstraction* of f in $\mathcal{A}$ iff $f^\alpha = \alpha \circ f \circ \gamma : A \to A$, because for any sound abstraction $f^\#$ it holds that $f^\alpha(a) \leq_A f^\#(a)$ for any $a \in A$.

Symbolic Abstraction. The above equation $f^\alpha = \alpha \circ f \circ \gamma$ defines the limit of precision obtainable using the abstract domain $\mathcal{A}$. However, the definition is non-constructive, in that it does not provide an algorithm for deriving the best transfer function.

In their seminal work, Reps et al. [2004] introduce a framework for computing f^α, which applies to a formula φ in a logic $\mathcal{L}$ and an abstract domain $\mathcal{A} = \langle A, \leq, \sqcup, \alpha, \gamma \rangle$. The formula φ encodes the concrete semantics, such as the concrete transformer for an instruction, basic block, or loop-free program fragment. The goal of symbolic abstraction is to find the strongest consequence of φ that is expressible in $\mathcal{A}$. More precisely,

Definition 2.1. (Symbolic Abstraction) Given a formula $\varphi \in \mathcal{L}$ and an abstract domain $\mathcal{A} = \langle A, \leq, \sqcup, \alpha, \gamma \rangle$, and let $[\![\varphi]\!]$ be the set of concrete states satisfying φ. The symbolic abstraction of φ in the domain $\mathcal{A}$, is an element $a \in A$ such that (1) a over-approximates the meaning of φ, i.e., $[\![\varphi]\!] \subseteq \gamma(a)$, and (2) for any $a' \in A$ for which $[\![\varphi]\!] \subseteq \gamma(a')$, we have $a \leq a'$.

Example 2.2. Consider the integer formula $\varphi(x, y) \equiv x \geq 0 \wedge y \geq 0 \wedge x + y \leq 10$ where x and y represent unbounded integers. The interval $x \in [0, +\infty] \wedge y \in [0, +\infty]$ is a sound approximation of φ in the interval domain, while $x \in [0, 10] \wedge y \in [0, 10]$ is the best approximation.

2.2 Optimization Modulo Theories

In this paper, we consider first-order logic formulas of the satisfiability modulo theories (SMT). Given a formula φ, we denote its free variables by $vars(\varphi)$. A model M of φ (denoted $M \models \varphi$) is a function that maps all free variables $x_1, \ldots, x_n \in vars(\varphi)$ to values in their respective domains such that φ evaluates to true. We denote $M(x)$ by the value of the variable x under the model M.

The problem of Optimization Modulo Theories (OMT) extends SMT by searching models that optimize some objective functions. Recall that a general mathematical optimization problem can be written as

$$\begin{cases} \text{maximize} & f(\bar{x}) \\ \text{subject to} & \bar{x} \in \mathcal{S}, \end{cases}$$

where $f(\bar{x})$ is the objective, and $\mathcal{S}$ the search space. In the context of OMT, $\mathcal{S}$ is characterized by a first-order formula φ in a background theory and $f(\bar{x})$ is a term of the theory.

Definition 2.3. (Boxed OMT Problem) Given an SMT formula φ and a set of objectives $\{g_1, \ldots, g_n\}$, the goal of the *multiple-independent-objective OMT problem* [Sebastiani and Trentin 2015b], a.k.a.

boxed OMT is to find a set of models $\{M_1, \ldots, M_n\}$ of φ such that each M_i maximizes the objective g_i respectively.

Previous work [Jiang et al. 2017; Li et al. 2014] has shown that the symbolic abstraction of template linear domains such as interval [Cousot and Cousot 1977], zone [Miné 2001], and octagon [Miné 2006] can be reduced to solving boxed OMT problems. Specifically, let e_i ($1 \leq i \leq n$, where n is the number of templates) be a template, we get c_i by solving the OMT problem "max e_i $s.t.$ φ", and thus obtain $e_i \leq c_i$ as a constraint in the template abstract domain representation. Overall, $\bigwedge_{i=1}^{n} e_i \leq c_i$ gives the resulting constraint representation in the template abstract domain. For example, by setting the template e as $vars(\varphi)$ and their negation, we can obtain the symbolic abstraction of φ in the interval domain.

Example 2.4. Consider the integer formula $\varphi(x, y) \equiv x \geq 0 \wedge y \geq 0 \wedge x + y \leq 10$ in Example 2.2. By setting the template as $\{x, y, -x, -y\}$ and solving the boxed OMT problem "max $\{x, y, -x, -y\}$ $s.t.$ φ", we can obtain the maximal/minimal values of x and y. Clearly, the symbolic abstraction of φ in the interval domain is $x \in [0, 10] \wedge y \in [0, 10]$.

2.3 Problem Formulation

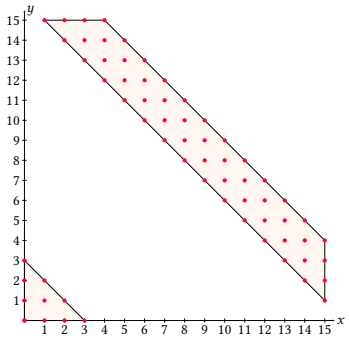
While OMT-based formalization offers an elegant solution for the symbolic abstraction of template linear domains, it faces several limitations in practice. First, it depends heavily on the performance of the underlying OMT solvers, and we observe that many real-world symbolic abstraction instances pose challenges to state-of-the-art solvers. Second, the formalization is not directly applicable to the convex polyhedral domain [Cousot and Halbwachs 1978], i.e., conjunctions of linear inequalities $a_1x_1 + \cdots + a_nx_n \leq c$, because neither is the number of templates known before the analysis, nor are the coefficients in each template.

This paper concerns the connections between Optimization Modulo Theories and symbolic abstraction. We focus on the theory of quantifier-free bit-vector (QF_BV), because it allows for modeling machine instructions faithfully and precisely, such as non-linear arithmetic computations and "bit-twiddling" operations (left-shift, right-shift; bitwise-and, bitwise-or, and bitwise-xor; etc.) [Alizadeh and Fujita 2009; Ganesh and Dill 2007; Lim and Reps 2013].

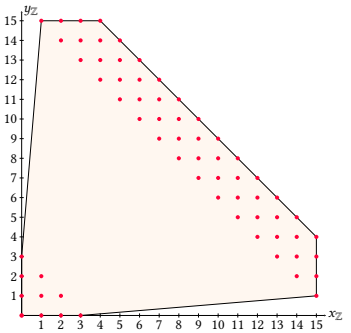
Problem Scope. Table 1 presents three abstract domains for bit-vector arithmetic, where v denotes variables and c_i denotes constants. Our work focuses on the first and third domains. The bit-vector interval domain is similar to the conventional interval domain, except that c_i and v are bit-vectors. However, it is important to understand the distinction between the two different but related polyhedral domains. Specifically, we introduce a new domain for bit-vector arithmetic, namely the *integral polyhedral domain*, which involves both integer and bit-vector reasoning. To illustrate, let us first consider the following example.

Example 2.5. Consider a bit-vector formula $\varphi \equiv x \geq 0 \wedge y \geq 0 \wedge x + y \leq 3$ where x and y encode two 4-bit unsigned integers. When $\varphi(x, y)$ is interpreted in bit-vector arithmetic, it has 64 models, shown as red points in Figure 1a. Note that $\varphi(x, y)$ is already a conjunction of three linear inequalities, and thus is its own symbolic abstraction as a bit-vector polyhedron: i.e., $\varphi' \equiv x \geq 0 \wedge y \geq 0 \wedge x + y \leq 10$ (equivalent to φ).

Now consider Figure 1b, which shows the same 64 models in red. These points can also be considered as 64 *integer* models, where each integral model corresponds to a bit-vector model of φ, and $x_{\mathbb{Z}}$ and $y_{\mathbb{Z}}$ are the integer variables corresponding to x and y, respectively. To over-approximate these 64 points using linear inequalities over integers, we need a conjunction of seven inequalities, as shown in Figure 1b, and listed explicitly in the caption of Figure 1b.



(a) bit-vector polyhedral abstraction $\varphi' \equiv x \geq 0 \wedge y \geq 0 \wedge x + y \leq 3$.



(b) Integral polyhedral abstraction $\varphi' \equiv 0 \leq x_{\mathbb{Z}} \leq 15 \wedge 0 \leq y_{\mathbb{Z}} \leq 15 \wedge y_{\mathbb{Z}} \leq 12 * x_{\mathbb{Z}} + 3 \wedge x_{\mathbb{Z}} \leq 12 * y_{\mathbb{Z}} + 3 \wedge x_{\mathbb{Z}} + y_{\mathbb{Z}} \leq 19$.

Fig. 1. Symbolic abstractions of the bit-vector formula $\varphi \equiv x \geq 0 \wedge y \geq 0 \wedge x + y \leq 3$ from Example 2.5.

In summary, the orange regions of Figure 1a and Figure 1b show the abstractions of φ in the bit-vector polyhedral domain and integral polyhedral domain, respectively. Observe that the orange region in Figure 1a (i.e., $\varphi' \equiv x \geq 0 \wedge y \geq 0 \wedge x + y \leq 3$) is non-convex.

The above example illustrates a key difference between the two polyhedral domains. Any element in the integral polyhedral domain must be a convex polyhedron (e.g., as the orange region in Figure 1b). However, this is not the case for the bit-vector polyhedral domain, because a conjunctive of bit-vector inequalities may describe a non-convex region (e.g., due to the presence of overflows [Sharma et al. 2013], as demonstrated in Figure 1a).

Now, to compare and contrast the two domains formally, we define a lifting operation for models of a bit-vector formula. Let $[\![\varphi]\!]_{bv}$ be the set of all models of a (satisfiable) bit-vector formula φ. Essentially, a bit-vector value is an integer modulo 2^w, i.e., an integer in $\mathbb{Z}_{2^w}$ for some bit width w. Thus, we can maintain and leverage a dual interpretation of bit-vector values.

Definition 2.6. (Integral Lifting of Bit-Vector Models) Let $M_{bv} \in [\![\varphi]\!]_{bv}$ be a model of a bit-vector formula φ, we say that

$$M_{int} = lift(M_{bv})$$

is the integral model lifted from a bit-vector model M_{bv}, by mapping each bit-vector variable $v \in vars(\varphi)$ to a unique integer variable $v_{\mathbb{Z}}$ and tracking the relations of their values.[2] We say that $[\![\varphi]\!]_{int}$ is the the set of all integral models that are lifted from $[\![\varphi]\!]_{bv}$, i.e.,

$$[\![\varphi]\!]_{int} = \{lift(M_{bv}) \mid \forall M_{bv} \in [\![\varphi]\!]_{bv}\}$$

Based on the definition above, we distinguish the corresponding two versions of symbolic polyhedral abstraction problem for a bit-vector formula φ:

- *Version 1*: A sound *bit-vector polyhedral abstraction* of φ is a conjunction of linear bit-vector formulas that cover all the models in $[\![\varphi]\!]_{bv}$. The best abstraction is a conjunction of linear bit-vector formulas ϕ for which (i) $[\![\phi]\!]_{bv} \supseteq [\![\varphi]\!]_{bv}$, and (ii) there does not exist a conjunction of linear bit-vector formula ψ such that $[\![\phi]\!]_{bv} \supsetneq [\![\psi]\!]_{bv} \supseteq [\![\varphi]\!]_{bv}$.

[2] For example, let v be a n-bits unsigned bit-vector variable and $v_{\mathbb{Z}}$ its corresponding integer variable. Suppose that in a bit-vector model, $b_{n-1}, \ldots, b_0$ are the values of the n bits in v. Then, the integer value of $v_{\mathbb{Z}}$ can computed as $b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \cdots + b_0 \times 2^0$.

Table 1. Three abstract domains for a bit-vector formula $\varphi \in$ QF_BV. Each domain can be regarded as an impoverished logic fragment $\mathcal{L}'$ (compare with the full QF_BV logic or linear integer arithmetic). An element $a \in \mathcal{A}$ can be represented as a formula $\varphi' \in \mathcal{L}'$.

Domain $\mathcal{A}$	Logic $\mathcal{L}'$	Interpretation
Bit-vector interval	inequalities of the form $c_1 \leq v$ and $v \leq c_2$	$[\![\varphi']\!] \supseteq [\![\varphi]\!](\varphi' \in \mathcal{L}')$
Bit-vector polyhedron	linear inequalities over bit-vectors	$[\![\varphi']\!] \supseteq [\![\varphi]\!](\varphi' \in \mathcal{L}')$
Integral polyhedron	linear inequalities over integers	Detailed below

- *Version 2*: A sound *integral polyhedral abstraction* of φ is a conjunction of linear integer formulas that cover all the models in $[\![\varphi]\!]_{int}$. The best abstraction is a conjunction of linear integer formulas ϕ for which (i) $[\![\phi]\!]_{int} \supseteq [\![\varphi]\!]_{int}$, and (ii) there does not exist a conjunction of linear integer formulas ψ such that $[\![\phi]\!]_{int} \supsetneq [\![\psi]\!]_{int} \supseteq [\![\varphi]\!]_{int}$.

In this paper, we address Version 2 of the problem. Recall that any sound solution to Version 2, including the optimal solution, must be a convex polyhedron (discussed in Example 2.5). The key benefit of this property is that it allows us to pass the computed polyhedron to existing algorithms for integer/real arithmetic, such as counting the number of models [Assarf et al. 2017], sampling solutions [Chen et al. 2018], computing the volume [Dyer and Frieze 1988], and computing the Hausdorff distance of two polyhedrons [Sankaranarayanan et al. 2006]. Such information is beneficial for many program analysis clients such as WCET analysis [Lisper 2003] and quantitative information flow [Biondi et al. 2018]. We will demonstrate one client in § 6.2.

Problem Statement. Based on the above discussion, our work aims to address the following challenges in symbolic abstraction: Given a (satisfiable) quantifier-free bit-vector formula φ,

> Challenge 1: Improve the performance of the OMT-based solution for computing the symbolic abstraction of φ in the bit-vector interval domain.

> Challenge 2: Lift OMT solving techniques to the polyhedral domain, and yield (more) efficient algorithms for the symbolic abstraction of φ in the *integral polyhedral domain*.

The key idea behind our work is that the variables in programs are often correlated, and the interval and polyhedral domains in the bit-vector arithmetic are bounded. The correlations and boundedness can be utilized to reduce redundant computations in symbolic abstraction. Specifically, the analysis method presented in the paper addresses these challenges by the following means:

- Challenge 1 is addressed via an SMT-based symbolic interval analysis that iteratively finds the optimal intervals for different variables synergistically, whereby the intermediate information computed for different variables is shared.
- Challenge 2 is addressed via a symbolic polyhedral analysis that interleaves the computations of interval and polyhedron. The key idea is to utilize the optimal interval abstractions to approach more "extremal" points near to the final integral polyhedron, which is bounded.

3 SYMBOLIC INTERVAL ABSTRACTION

In this section, we introduce our approach to symbolic interval abstraction. Without loss of generality, we formalize the problem as a boxed multi-objective optimization instance: given a first-order formula φ and a set of objectives $\{g_1, \ldots, g_n\}$, compute the objectives' maximal values.[3] For ease of presentation, we use unsigned bit-vectors to demonstrate our approach.

[3]Minimization problems can be reduced to maximization problems.

Algorithm 1: SMT-based binary search for optimizing a single objective.

```
   Input: A QF_BV formula φ and an objective g
   Output: The maximum value of g s.t. φ
1  Function optimize_one_obj(φ, g)
2      ret, low, high ← ...;
3      while low ≤ high do
4          mid ← (low + high)/2;
5          ψ ← φ ∧ (mid ≤ g ≤ high);
6          if ψ is unsatisfiable then
7              high ← mid − 1;
8          else
9              M ← a model of ψ;                    /* use M to update ret and low */
10             ret ← M(g), low ← ret + 1;
11     return ret;
```

At a high level, our procedure consists of two steps. First, we leverage a sound and lightweight interval analysis [Gange et al. 2015] to compute the initial abstractions of the variables. Second, we perform a synergistic SMT-based refinement until finding all the maximum values of the variables. In the following, we focus on illustrating the second step, which is the key to improving performance while ensuring optimality.

3.1 Basic Binary Search

Our approach builds on the standard binary search schema. Algorithm 1 shows the basic procedure for maximizing a single objective, which takes as input an initial lower bound and upper bound (denoted "*low*" and "*high*" respectively). Since we target fixed-sized bit-vector formulas, any variable is guaranteed to be bounded. The algorithm iteratively updates "*low*" and "*high*" using the satisfiability results of $\varphi \wedge mid \leq g \leq high$ (Lines 6-10), until the value of *low* is larger than *high*.[4] Assume that the variable g encodes an m-bits unsigned integer. In the worst case, Algorithm 1 needs to call an SMT solver m times.

Example 3.1. Consider a bit-vector formula $\varphi(x)$ where x encodes a 3-bits unsigned integer. On the first round of a binary search, we have $low = 0$, $high = 7$, and $mid = 4$. Thus, Algorithm 1 needs to solve the formula $\varphi \wedge 4 \leq x \leq 7$.

We remark that Algorithm 1 is similar to many other existing algorithms [Henry et al. 2014; Köksal et al. 2012; Nadel and Ryvchin 2016; Sebastiani and Tomasi 2015a] that maximize/minimize one objective subject to a formula. The focus of our symbolic interval analysis is to accelerate the computation of multiple variables. Suppose we need to maximize n variables of a formula, where each variable encodes an m-bits unsigned integer. A naive solution to the problem is Algorithm 2, which invokes Algorithm 1 for n times. However, if the value of n is large, such a strategy can suffer from performance issues, where little information is shared among different variables.

[4]Note that in Algorithm 1, we should replace $mid = (low + high)/2$ by $mid = low + (high - low)/2$ to avoid computing incorrect mid-points due to overflow. We show the previous one for simplicity.

Algorithm 2: Naive SMT-based binary search for optimizing multiple objectives.

Input: A QF_BV formula φ and a set of objectives $G = \{g_1, \ldots, g_n\}$
Output: The maximum values of $g_1, \ldots, g_n$ *s.t.* φ

```
1  Function optimize_multi_obj(φ, G)
2      ret_1, ..., ret_n ← ...;
3      foreach g_i ∈ G do
4          ret_i ← optimize_one_obj(φ, g_i);        /* invoke Algorithm 1 */
5      return ret_1, ..., ret_n;
```

Algorithm 3: Solving the conjunctive predicate abstraction problem.

Input: A formula φ and a set of predicates $S = \{\phi_1, \ldots, \phi_n\}$
Output: Decide the satisfiability of each $\varphi \wedge \phi_i$ $(1 \leq i \leq n)$

```
1  Function decide_cpa(φ, S)
2      while S ≠ ∅ do
3          Ψ ← ⋁_{φ_i ∈ S} φ_i;                    /* merge the predicates */
4          if φ ∧ Ψ is unsatisfiable then
5              foreach φ_i ∈ S do
6                  mark φ ∧ φ_i as unsatisfiable;
7                  return;
8          else
9              M ← a model of φ ∧ Ψ;                /* use M to filter φ_i */
10             foreach φ_i ∈ S do
11                 if M ⊨ φ_i then
12                     mark φ ∧ φ_i as satisfiable;
13                     remove φ_i from S;
```

3.2 Factorizing the Search

Our idea for improving the performance of multi-objective optimization is to handle the variables simultaneously, during which we reuse information computed for different variables. Recall that when dealing with a set of objectives, Algorithm 2 needs to solve a number of SMT queries $\{\varphi \wedge t_1, \ldots, \varphi \wedge t_n\}$, where t_i is of the form $mid_i \leq g_i \leq high_i$. Essentially, we are dealing with the following question, which we term as the problem of "conjunctive predicate abstraction":

> Given a formula φ and a set of predicates $S = \{\phi_1, \ldots, \phi_n\}$, decide for each $\phi_i \in S$, if $\varphi \wedge \phi_i$ is satisfiable or not.

The problem is actually prevalent in program analysis and verification tasks. Simply feeding each $\varphi \wedge \phi_i$ to the solver can be inefficient, if the size of S is large and there are many instances of such queries. To potentially reduce the number of SMT solver calls, we utilize Algorithm 3 to solve such problems. The algorithm repeats the following steps until all predicates in S are decided.

(1) First, we construct a formula Ψ (Line 3) as the disjunction of all undecided predicates in S. Clearly, $\varphi \wedge \Psi$ is an over-approximation of each $\varphi \wedge \phi_i$.

(2) If $\varphi \wedge \Psi$ is unsatisfiable, then for all $\phi_i \in S$, we have that $\varphi \wedge \phi_i$ is unsatisfiable (Line 6). Algorithm 3 can terminate here.
(3) Else, we extract a model M of $\varphi \wedge \Psi$, and test if $M \models \phi_i$ for each ϕ_i. If this is the case, we can mark $\varphi \wedge \phi_i$ as satisfiable and remove ϕ_i from S for further consideration (Lines 9-13).

Note that, we do not have to start the solver from scratch in each iteration, by utilizing the incremental solving techniques (such as phase saving and clause learning) in modern SMT solvers.

Example 3.2. Consider a bit-vector formula $\varphi \equiv x \leq 2 \wedge \cdots \wedge y \leq 3$ where x and y encode two 3-bits unsigned integers. At the first round of the binary search, we need to decide the satisfiability of $\varphi \wedge 4 \leq x \leq 7$ and $\varphi \wedge 4 \leq y \leq 7$, respectively. Using Algorithm 3, we construct a formula $\varphi \wedge (4 \leq x \leq 7 \vee 4 \leq y \leq 7)$, which is unsatisfiable. Thus, we have that both $\varphi \wedge 4 \leq x \leq 7$ and $\varphi \wedge 4 \leq y \leq 7$ are unsatisfiable.

Proposition 1. Assuming k out of the n formulas in S can be satisfied in conjunction with φ, then Algorithm 3 needs at most $min(k+1, n)$ times of SMT calls.

Proof. (1) Assuming $k + 1 \leq n$. First, deciding the $n - k$ unsatisfiable formulas needs only the last SMT call. Second, at each round before the last iteration, at least one formula can be decided. Thus, the worst-case number of SMT calls is $k/1 + 1 = k + 1$. (2) Assuming $k + 1 > n$ (i.e., $k = n$). Since at least one formula can be decided after every SMT call, the worst-case number of SMT calls is $n/1 = n$. Taken (1) and (2) together, we conclude that Algorithm 3 needs at most $min(k + 1, n)$ times of SMT calls. □

3.3 Putting It All Together

Algorithm 4 shows the overall procedure to optimizing multiple objectives. Given a quantifier-free bit-vector formula φ and a set of objectives G to maximize, it first extracts an initial abstraction with a sound interval analysis [Gange et al. 2015] (Line 2), and then performs the SMT-based binary search to obtain the optimal values (Lines 3-13).

The key idea behind Algorithm 4 is to reuse information computed for one variable in order to speed up the optimization of other variables. This feature allows us to reduce the search space and avoid repeating expensive SMT calls. For instance, if a model M_1 indicates a maximal value for x, then a model M_2 for a maximal value of y must also satisfy $M_2(y) \geq M_1(y)$. Thus, we can possibly update both $high_x$ and $high_y$ within one SMT solver call.

Specifically, Algorithm 4 utilizes the sub-procedure `decide_cpa_ext` to enable sharing information among the variables (Lines 14-27). The sub-procedure is essentially an extension of Algorithm 3. Observe that `decide_cpa_ext` differs from Algorithm 3, in that it can update elements of S within the iteration (Lines 24-27). This is because in the binary-search-based algorithm, we can update low_i immediately, once knowing that $\varphi \wedge mid_i \leq g_i \leq high_i$ is satisfiable.

Proposition 2. Given a formula φ and n variables, each of which represents an m-bits unsigned integer. The worst-case number of SMT calls required by Algorithm 4 is $n * m$.

Although the number of SMT calls can still be huge, we summarize a few properties of Algorithm 4. First, the sound interval analysis in the first step can possibly reduce the value of m for some variables. Second, during the second step, Algorithm 4 can update the information of 1 to n variables with every SMT call. Third, when running out of a time budget, the algorithm may still retrieve the optimal values for a subset of the variables. Specifically, during the binary search, it can check if $low_i > high_i$ to decide if g_i has been optimized.

Example 3.3. Consider a bit-vector formula $\varphi \equiv x \geq 0 \wedge y \geq 0 \wedge x + y \leq 3$ where x and y encode two 4-bit unsigned integers. Note that the bit-vector addition $x + y$ may overflow. Figure 2a and

Algorithm 4: Optimized boxed multi-objective optimization.

Input: A QF_BV formula φ and a set of objectives $G = \{g_1, \ldots, g_n\}$
Output: The maximal values of $g_1, \ldots, g_n$ *s.t.* φ

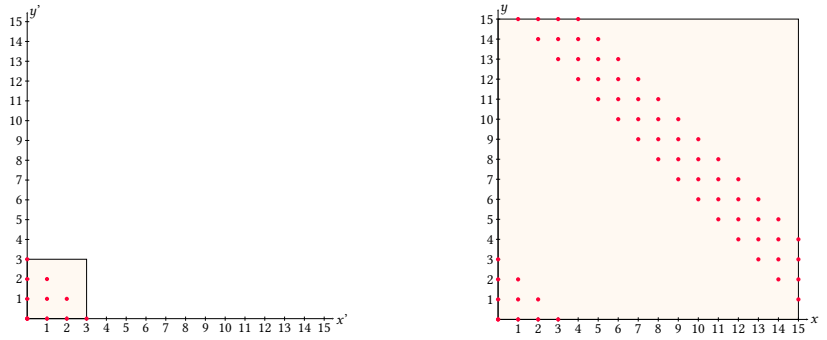
```
1  Function optimize_multi_obj(φ, G)
2      initialize low_i, high_i, ret_i with an interval analysis [Gange et al. 2015];
3      while true do
4          S ← ∅;
5          foreach g_i ∈ G do
6              if low_i ≤ high_i then
7                  mid_i ← (low_i + high_i)/2;
8                  S ← S ∪ {mid_i ≤ g_i ≤ high_i};
9          if S == ∅ then
10             break;                                  /* all variables optimized */
11         else
12             decide_cpa_ext(φ, S);                   /* an extension of Algorithm 3 */
13     return ret_1, ..., ret_n;
14 Function decide_cpa_ext(φ, S):
15     while true do
16         Ψ ← ⋁_{φ_i ∈ S} φ_i;                        /* merge the predicates */
17         if φ ∧ Ψ is unsatisfiable then
18             foreach φ_i ∈ S do
19                 high_i ← mid_i − 1;
20             return;
21         else
22             M ← a model of φ ∧ Ψ;                   /* use M to update low_i and mid_i */
23             foreach φ_i ∈ S do
24                 if M ⊨ φ_i then
25                     ret_i ← M(g_i), low_i ← ret_i + 1;
26                     mid_i ← (low_i + high_i)/2;
27                     φ_i ← mid_i ≤ g_i ≤ high_i;
```

Figure 2b demonstrate the effects of overflows on our algorithm. The red points depict the models of the formula. The orange regions show the results of symbolic interval abstractions. Specifically, if considering overflows, φ can have some "additional models" such as $(15, 1)$ and $(1, 15)$, as shown in Figure 2b. Consequently, the symbolic interval abstraction of φ is $x \in [0, 15] \wedge y \in [0, 15]$.

By default, we consider the overflow semantics. Thus, our algorithm can compute the abstraction as $x \in [0, 15] \wedge y \in [0, 15]$ (Figure 2b). We will discuss more about the handling of overflow in § 5.

4 SYMBOLIC POLYHEDRAL ABSTRACTION

In this section, we introduce our approach to symbolic polyhedral abstraction, which builds on the RSY algorithm [Reps et al. 2004] and uses the interval abstraction algorithm presented in § 3 as a



(a) In the absence of overflow. (b) In the presence of overflow.

Fig. 2. Symbolic interval abstraction of the bit-vector formula $\varphi \equiv x \geq 0 \wedge y \geq 0 \wedge x + y \leq 3$. The red points are the models. The orange regions are the symbolic interval abstractions.

Algorithm 5: RSY algorithm for symbolic abstraction [Reps et al. 2004].

Input: A formula φ and an abstract domain $\mathcal{A}$
Output: The symbolic abstraction a of φ in $\mathcal{A}$

```
1 Function rsy(φ, A)
2     ψ ← φ, a ← ⊥;
3     while ψ is satisfiable do
4         M ← a model of ψ;
5         a ← a ⊔ α(M);   /* use the model M to update the current abstraction */
6         ψ ← ψ ∧ ¬γ(a); /* bock the models covered by the updated abstraction */
7     return a;
```

sub-procedure. We first illustrate the RSY algorithm and use an example to motivate our algorithm. We then introduce our algorithms that interleave the computations of intervals and polyhedrons.

Notice. In this section, we assume dealing with bit-vectors with a signed interpretation. The symbolic interval analysis algorithm (§ 3) can easily be extended to support signed bit-vectors.

4.1 The RSY Algorithm

Algorithm 5 presents the Reps, Sagiv, and Yorsh (RSY) [Reps et al. 2004]'s parametric approach to symbolic abstraction. It takes as input a formula φ and an abstract domain $\mathcal{A}$, and yields the most precise element $a \in \mathcal{A}$ that over-approximates the concrete states described by φ. The algorithm keeps a lower bound *ret* of the correct result, and iteratively refines *ret* until it is no longer an under-approximation of the result. More specifically, each iteration of the algorithm has two steps:

- *Sampling*: this step invokes the decision procedure to generate a model of φ (Line 4).
- *Generalization*: this step generalizes the current abstraction of φ with the sampled model (Line 5), and adds the blocking formula to bock the models that can be covered by the updated abstraction (Line 6).

Essentially, Algorithm 5 starts from the lattice $\bot$ to the lattice $\top$. The algorithm iteratively enumerates and generalizes the models of a formula φ, until the abstraction "encompasses" all models of φ. In the worst case, Algorithm 5 requires h calls to a decision procedure, where h is the chain-length of the respective abstract domain $\mathcal{A}$.

Example 4.1. Consider a bit-vector formula $\varphi(x)$ where x encodes an 8-bit signed integer. Suppose that φ has the following set of 32 models:

$$\{([\![x]\!] = -1), ([\![x]\!] = 0), ([\![x]\!] = 1), \ldots, ([\![x]\!] = 30)\} .$$

The symbolic interval abstraction of formula is $x \in [-1, 30]$. To motivate our approach, let us run Algorithm 5 on the formula for the interval domain. At the first iteration, assume that the SMT solver yields a model $M_1 = ([\![x]\!] = -1)$, which induces $-1 \leq x \leq -1$. At the next iteration, the formula $\varphi \wedge \neg(-1 \leq x \leq -1)$ is passed to the solver, possibly yielding a model $M_2 = ([\![x]\!] = 0)$ that defines a constraint $-1 \leq x \leq 0$. Proceeding as before, the algorithm requires 32 solver calls to converge onto the final abstraction $-1 \leq x \leq 30$.

However, Algorithm 5 could have computed the abstraction $-1 \leq x \leq 30$ in three iterations. assuming that the SMT solver returned models $M_1 = ([\![x]\!] = -1)$ and $M_2 = ([\![x]\!] = 30)$ that define $x = -1$ and $x = 30$, respectively. At the last iteration, the SMT solver is used to prove that $\varphi \wedge \neg(-1 \leq x \leq 30)$ is unsatisfiable, i.e., $\varphi \models -1 \leq x \leq 30$.

As illustrated in the example above, Algorithm 5 needs to iteratively sample the models of the formula φ. The specific models yielded by the decision procedure affect the efficiency of Algorithm 5 largely, as they determine the search direction of the algorithm.

4.2 Our Approach

On Mixing Integer and Bit-vector Reasoning. Recall that we aim to compute the integral polyhedral abstraction of a bit-vector formula φ (§ 2.3), and our approach builds on the RSY algorithm. A difficulty is that the problem would involve the mixing of bit-vector and integer reasoning. On the one hand, we need to iteratively sample models of the bit-vector formula. On the other hand, we need to compute the convex hull of a set of integer models, which can be represented as a conjunction of linear integer formulas. The bit-vector and integer constraints would "interfere with each other" in a RSY-style algorithm. Thus, solving the problem necessitates a mechanism for (1) (model-theoretically) communicating the information between $[\![\varphi]\!]_{bv}$ and their integral lifting $[\![\varphi]\!]_{int}$, and (2) (proof-theoretically) mixing the reasoning of formulas in the bit-vector and linear integer theories.

As sketched in § 2.3, the idea underlying our solution is to maintain a dual interpretation of bit-vector values. Conceptually, we can create an integer variable $v_{\mathbb{Z}}$ for each bit-vector variable v, which allows us to convert models between $[\![\varphi]\!]_{bv}$ and $[\![\varphi]\!]_{int}$, and express both bit-vector and integer constraints. The relation between v and $v_{\mathbb{Z}}$ can be tracked in many different ways. In our implementation, we use the *bv2int* function supported by the Z3 SMT solver, by explicitly creating an extra constraint $v_{\mathbb{Z}} = bv2int(v)$.[5] By doing so, we can encode polyhedral constraints using $v_{\mathbb{Z}}$, and use the solver to sample models in $[\![\varphi]\!]_{bv}$ and $[\![\varphi]\!]_{int}$ simultaneously.

Abstraction from Intervals. Algorithm 6 outlines our first attempt, which takes as input a formula φ and initializes the polyhedral abstraction to $\bot$. The key idea behind the algorithm is to find "extremal" models that represent vertex closer to the final convex polyhedron. Compared with the temporal polyhedrons generalized with an arbitrary model, the polyhedrons induced by the extremal models can possibly cover more models of the formula, thereby speeding up the convergence of symbolic abstraction.

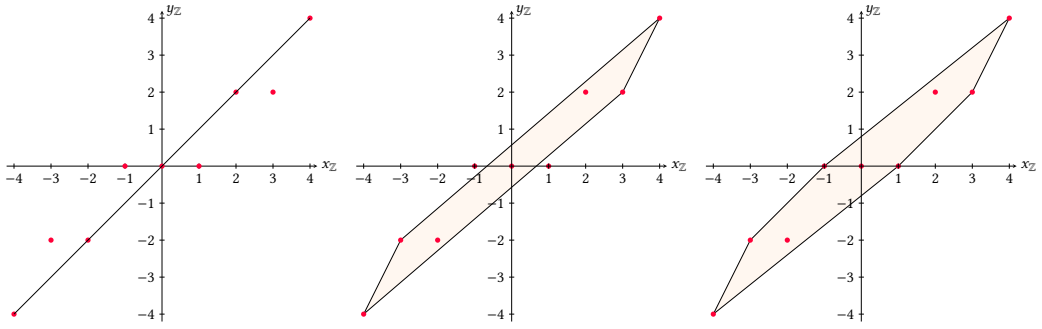
However, obtaining such models is non-trivial because the forms of the inequalities in the polyhedral abstraction are not known prior. The solution of Algorithm 6 is to utilize symbolic interval abstractions to sample the "extremal" models. More specifically, we compute the interval

[5] In the SMT-LIB2 standard, the function is called "bv2nat". For interested readers, we refer to http://smtlib.cs.uiowa.edu/theories-FixedSizeBitVectors.shtml.

Algorithm 6: Polyhedral abstraction with symbolic intervals.

Input: A QF_BV formula φ with n variables
Output: The symbolic polyhedral abstraction of φ

1 **Function** `polyhedral_abs_from_interval`(φ, S)
2 **foreach** $v \in vars(\varphi)$ **do**
3 $\varphi \leftarrow \varphi \wedge v_{\mathbb{Z}} = bv2int(v)$; `/* maintain a dual interpretation of` φ`'s models */`
4 $p \leftarrow \bot$;
5 **while** $\varphi \wedge \neg p$ is satisfiable **do**
6 $c \leftarrow \emptyset$;
7 **foreach** $v \in vars(\varphi)$ **do**
8 $[l, u] \leftarrow$ symbolic interval abstraction of v *s.t.* $\varphi \wedge \neg p$;
9 $M_l \leftarrow$ the model that maximizes v;
10 $M_u \leftarrow$ the model that minimizes v;
11 $c \leftarrow c \cup \{(M_l(v_{1_{\mathbb{Z}}}), \ldots, M_l(v_{n_{\mathbb{Z}}})), (M_u(v_{1_{\mathbb{Z}}}), \ldots, M_u(v_{n_{\mathbb{Z}}}))\}$;
12 $p \leftarrow p \sqcup \alpha(c)$; `/* update the current abstraction via polyhedral join */`
13 **return** p;



(a) p_1 after the first iteration. (b) p_2 after the second iteration. (c) p_3 after the third iteration.

Fig. 3. Major steps for computing the integral polyhedral abstraction of a bit-vector formula φ that has 9 models $\{(4,4),(3,2),(2,2),(1,0),(0,0),(-1,0),(-2,-2),(-3,-2),(-4,-4)\}$.

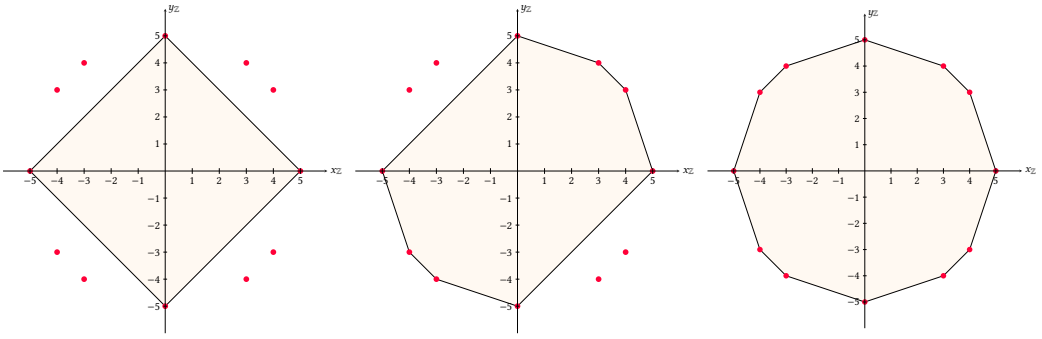
abstractions of $vars(\varphi)$ subject to $\varphi \wedge \neg p$, and utilize the models under which $vars(\varphi)$ are minimal or maximal (Lines 8-10). Then, the extremal models are collected in a set c, which is used to update the abstraction p via polyhedral join (Line 12). In summary, Algorithm 6 utilizes interval abstractions to control the distribution of the sampled models, instead of enumerating arbitrary points (as in Algorithm 5 [Reps et al. 2004]).

However, Algorithm 6 has a major obstacle to scalability, namely the interval abstractions (Lines 8). To speed up the computations, we leverage the interval abstraction algorithm introduced in § 3, which optimizes the variables simultaneously (the inner loops in Lines 7-11).

Example 4.2. Consider a bit-vector formula φ where x and y encode signed integers. Suppose that the set of models $[\![\varphi]\!]_{int}$ are depicted as the red points in Figure 3, which include

$$\{(4,4),(3,2),(2,2),(1,0),(0,0),(-1,0),(-2,-2),(-3,-2),(-4,-4)\}$$

Next, we run Algorithm 6 to compute the integral polyhedral abstraction of φ.



(a) p_1 after the first iteration. (b) p_2 after the first iteration. (c) p_3 after the third iteration.

Fig. 4. A bit-vector formula $\varphi \equiv -5 \leq x \leq 5 \wedge -5 \leq y \leq 5 \wedge x^2 + y^2 = 25$ that has 12 models, and the major steps to computing its integral polyhedral abstraction.

At the first iteration, we compute the symbolic interval abstractions of x and y subject to φ and obtain two integral models, $M_1 = (\llbracket x_{\mathbb{Z}} \rrbracket = 4, \llbracket y_{\mathbb{Z}} \rrbracket = 4)$ and $M_2 = (\llbracket x_{\mathbb{Z}} \rrbracket = -4, \llbracket y_{\mathbb{Z}} \rrbracket = -4)$, because x and y have maximum/minimum values under their corresponding bit-vector models.[6] The polyhedral abstraction from M_1 and M_2 are defined as below (see Figure 3a):

$$p_1 = \alpha(M_1(x_{\mathbb{Z}}), M_1(y_{\mathbb{Z}}), M_2(x_{\mathbb{Z}}), M_2(y_{\mathbb{Z}})) = \begin{cases} y_{\mathbb{Z}} \geq -4 \ \wedge \\ y_{\mathbb{Z}} \leq 4 \ \wedge \\ y_{\mathbb{Z}} = x_{\mathbb{Z}} \end{cases} \tag{1}$$

At the second iteration, we start with $\varphi \wedge \neg p_1$ and obtain two models $M_3 = (\llbracket x_{\mathbb{Z}} \rrbracket = 3, \llbracket y_{\mathbb{Z}} \rrbracket = 2)$ and $M_4 = (\llbracket x_{\mathbb{Z}} \rrbracket = -3, \llbracket y_{\mathbb{Z}} \rrbracket = -2)$. The convex hull of the models is joined with p_1, yielding the following polyhedron (see Figure 3b):

$$p_2 = \begin{cases} y_{\mathbb{Z}} \leq 2 * x_{\mathbb{Z}} + 4 \wedge y_{\mathbb{Z}} \geq 2 * x_{\mathbb{Z}} - 4 \ \wedge \\ 7 * y_{\mathbb{Z}} \leq 6 * x_{\mathbb{Z}} + 6 \wedge 7 * y_{\mathbb{Z}} \geq 6 * x_{\mathbb{Z}} - 4 \end{cases} \tag{2}$$

At the third iteration, we start with $\varphi \wedge \neg p_2$, obtaining two models $M_5 = (\llbracket x_{\mathbb{Z}} \rrbracket = 1, \llbracket y_{\mathbb{Z}} \rrbracket = 0)$ and $M_6 = (\llbracket x_{\mathbb{Z}} \rrbracket = -1, \llbracket y_{\mathbb{Z}} \rrbracket = 0)$. Similarly, we abstract M_5 and M_6 as a convex polyhedron and then join the polyhedron with p_2, yielding the following polyhedron (see Figure 3c)

$$p_3 = \begin{cases} y_{\mathbb{Z}} \leq 2 * x_{\mathbb{Z}} + 4 \wedge y_{\mathbb{Z}} \geq 2 * x_{\mathbb{Z}} - 4 \ \wedge \\ y_{\mathbb{Z}} \leq x_{\mathbb{Z}} + 1 \wedge 5 * y_{\mathbb{Z}} \geq 4 * x_{\mathbb{Z}} - 4 \ \wedge \\ 5 * y_{\mathbb{Z}} \leq 4 * x_{\mathbb{Z}} + 4 \wedge y_{\mathbb{Z}} \geq x_{\mathbb{Z}} - 1 \end{cases} \tag{3}$$

Finally, we find that $\varphi \wedge \neg p_3$ is unsatisfiable (i.e., $\varphi \models p_3$) and, thus, Algorithm 6 terminates with a sound abstraction. As shown in Figure 3c, all models in $\llbracket \varphi \rrbracket_{int}$ (i.e., the integral lifting of $\llbracket \varphi \rrbracket_{bv}$) have been encompassed by the final abstraction p_3.

Interleaving Algorithm 6 and RSY. Algorithm 6 utilizes symbolic interval abstractions to sample "extremal" models, aiming to converge within fewer iterations. However, obtaining an optimal polyhedral abstraction via Algorithm 6 may still be prohibitively expensive. For example,

[6]Note that in theory, the inner loops in Lines 7-11 (Algorithm 6) need to compute four models of φ, if it optimizes each variable one by one. In practice, we only compute two as we can optimize different variables simultaneously. For example, the bit-vector model corresponding to $M_1 = (\llbracket x_{\mathbb{Z}} \rrbracket = 4, \llbracket y_{\mathbb{Z}} \rrbracket = 4)$ can both maximize x and y.

Algorithm 7: Polyhedral abstraction by interlving RSY and Algorithm 6.

Input: A QF_BV formula φ
Output: The symbolic polyhedral abstraction of φ

Function `polyhedral_abs(`φ`)`
  foreach $v \in vars(\varphi)$ **do**
    $\varphi \leftarrow \varphi \wedge v_{\mathbb{Z}} = bv2int(v)$;
  $p \leftarrow \bot$;
  while $\varphi \wedge \neg p$ is satisfiable **do**
    $c \leftarrow \emptyset$;
    if *round_robin()* **then**
      ; `/* same as Algorithm 6 */`
      for $v \in vars(\varphi)$ **do**
        $[l, u] \leftarrow$ symbolic interval abstraction of v $s.t.$ $\varphi \wedge \neg p$;
        $M_l \leftarrow$ the model that maximizes v;
        $M_u \leftarrow$ the model that minimizes v;
        $c \leftarrow c \cup \{(M_l(v_{1_{\mathbb{Z}}}), \ldots, M_l(v_{n_{\mathbb{Z}}})), (M_u(v_{1_{\mathbb{Z}}}), \ldots, M_u(v_{n_{\mathbb{Z}}}))\}$;
      $p \leftarrow p \sqcup \alpha(c)$;
    else
      ; `/* the RSY mode (Algorithm 5) */`
      $M \leftarrow$ a model of $\varphi \wedge \neg p$;
      $c \leftarrow c \cup \{(M(v_{1_{\mathbb{Z}}}), \ldots, M(v_{n_{\mathbb{Z}}}))\}$;
      $p \leftarrow p \sqcup \alpha(c)$;
  return p;

consider a bit-vector formula $\varphi \equiv -5 \leq x \leq 5 \wedge x^2 + y^2 = 25$. Figure 4 shows $[\![\varphi]\!]_{int}$ that consists of 12 models:

$$\{(5,0),(4,3),(3,4),(0,5),(-3,4),(-4,3),(-5,0),(-4,-3),(-3,-4),(0,-5),\ (3,-4),\ (4,-3)\}$$

As shown in the figure, the models characterize a Dodecagon. Algorithm 6 can finish computing the best polyhedral abstraction in four iterations (Figures 4a-4c depict the results after the first, second, and third iterations). As can be seen, if running Algorithm 6 on the instance, we end up enumerating all the points in the Dodecagon using the (expensive) interval abstraction algorithm. Such a strategy can be very costly, if the number of models/points in the final polyhedron is huge.

Before presenting our final algorithm, let us consider running the RSY algorithm (Algorithm 5) on the formula in Figure 4c. Any model given by the SMT solver yields a vertex of the final convex polyhedron, because all models are vertices of the Dodecagon. As such, the intermediate models computed by the RSY algorithm have similar effects to the ones in Algorithm 6. However, for other formulas with many non-extremal models, RSY can end up sampling non-extremal points, and require many rounds before converging to the desired answer.

To summarize, the RSY algorithm samples one model with each SMT call, but the model may not generalize well. Algorithm 6 uses symbolic interval abstractions to sample "extremal" models, which generalize well but can be hard to compute. Therefore, our final solution, Algorithm 7, interleaves the original RSY algorithm (Algorithm 5) with the interval-abstraction-based one (Algorithm 6), which aims to balance the cost of sampling the models and the "quality" of the sampled models. In

our client, we have empirically experienced the best performance with one interval-abstraction step after every k RSY iterations, always starting the search in the RSY mode. [7]

Proposition 3. There exists a best integral polyhedral abstraction for any bit-vector formula φ, which can be computed by Algorithms 5–7 (in the absence of timeouts). [8]

Remarks. To compute the best abstraction of φ, a naive strategy is to first enumerate all models in $[\![\varphi]\!]_{int}$, and then compute their convex hull. Essentially, Algorithms 5-7 optimize the naive strategy by avoiding explicitly enumerating all the models, and by controlling the distributions of the sampled models.

5 ON OVERFLOW AND UNDERFLOW

An important and long-standing challenge in designing numeric domains is to to soundly track the effects of arithmetic operations in machine integers, such as the wrap-around effects of operations that overflow [Blanchet et al. 2003; Bygde et al. 2012; Cousot and Halbwachs 1978; Gange et al. 2015; Sharma et al. 2013; Sharma and Reps 2017; Simon and King 2007]. To tame the complexity, the quantifier-free bit-vector theory (QF_BV) allows for faithfully modeling machine integer semantics, such as bit-wise operations, overflow, underflow, and others.

Our work aims to compute the symbolic abstraction of QF_BV formulas and can account for the machine integer semantics such as overflow and underflow. Take the handling of overflow as an example. First, the presence of overflow does not break the assumption that the value range of a bit vector is always finite, which is a key to our algorithms. Second, our algorithms leverage the SMT solver to sample the models of a formula, which can return models caused by overflow. The algorithms do not attempt to prune those models.

An Encoding Schema for Preventing Overflow/Underflow. However, as illustrated in Example 3.3 (§ 3.3), the presence of overflow or underflow can affect the results of symbolic abstraction greatly. Previous work [Ritter 2015] shows that for many applications such as compiler optimization, it is practical to assume the absence of certain undefined behavior like signed integer overflow. Thus, given a bit-vector formula φ to abstract, it is desirable to allow for controlling the overflow and/or underflow in the arithmetic computations in φ. In what follows, we present an encoding schema for such application scenarios, using the handling of overflow as an example.

Consider the formula $\varphi \equiv y = 2 + x \wedge y > 10$, where x and y encode signed 32-bits integers. In practice, there are many ways to implement the assumption that a bit-vector operation such as $2 + x$ does not overflow. In our implementation, we have used a series of Z3 APIs, such as `bvadd_no_overflow` and `bvadd_no_overflow`,[9] to ensure that an SMT solver does not produce models caused by overflow. Specifically, we can rewrite φ as:

$$\varphi' \equiv y = 2 + x \wedge b = \texttt{bvadd\_no\_overflow}(2 + x) \wedge y > 10$$

where the function $\texttt{bvadd\_no\_overflow}(2 + x)$ returns a Boolean-typed value indicating whether the bit-vector addition leads to an overflow. That is, the variable b is true if and only if $2 + x$ does not overflow. Now, if we need to enforce the absence of overflow, we can add a further conjunction with b, which yields:

$$\varphi'' \equiv y = 2 + x \wedge b = \texttt{bvadd\_no\_overflow}(2 + x) \wedge y > 10 \wedge b$$

[7] In our experiments, we set k as 10 for the polyhedral abstraction queries.

[8] The proof is given in the extended version of the paper at https://tinyurl.com/w6ck5uub.

[9] The APIs are essentially wrappers of the standard bit-vector operations. For example, to track overflows in a bit-vector addition, we can just zero-extend bit-vector arguments by one. Then, we check whether the most significant bit in the output is 0. See https://github.com/Z3Prover/z3/blob/master/src/api/api_bv.cpp for more implementation details.

Note that the above schema can also be used to enforce different modes (regular, underflow, overflow) for different instructions. For example, we can prevent overflows for the subset of instructions that involve signed integers, but not for unsigned ones.

6 IMPLEMENTATION AND APPLICATIONS

We have implemented our approach as a tool called Taichi, using Z3 [De Moura and Bjørner 2008] as the SMT solver. Taichi treats the underlying SMT solver as a black box. This makes it easy to implement and allows it to benefit from future advances in SMT solving. We term the analyses computing interval and polyhedral abstractions as TaichiInt and TaichiPoly, respectively. We have applied TaichiInt and TaichiPoly in two machine code analysis clients, namely static vulnerabilities detection and dynamic program testing, respectively.

6.1 Memory Corruption Analysis with TaichiInt

The first application is to enhance an abstract interpreter, which implements a memory corruption analysis on top of the Angr binary analysis platform [Shoshitaishvili et al. 2016]. The implemented analysis uses Angr to translate the binary code to VEX, the intermediate representation used by Valgrind [Nethercote and Seward 2007]. We then perform both the conventional interval analysis and the symbolic interval abstraction on top of the VEX intermediate representation.

At the core of the memory corruption analysis is a conventional statement-by-statement interval analysis. The analysis has a high false-positive rate due to a lack of accuracy in the interval analysis. For example, aliases of two memory accesses are determined by checking if their intervals share common address values. This is similar to the value set analysis [Reps and Balakrishnan 2008]. The imprecision of the interval information can result in many spurious aliasing relations.

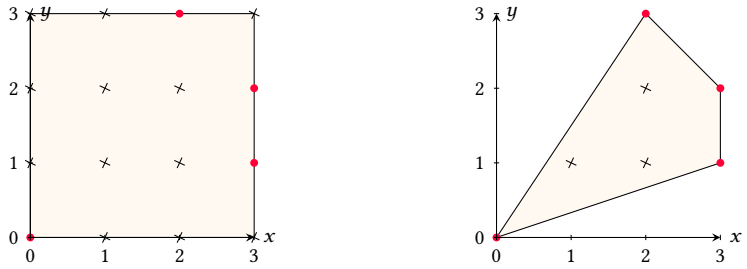
As a remedy, we apply symbolic abstraction in a demand-driven style. If the conventional interval analysis fails to verify some functions' memory safety, we use TaichiInt to obtain a possibly more precise interval. At a high level, we partition a program into several "SMT-expressible" and loop-free blocks (as in "large block encoding" [Beyer et al. 2009]), and compute the fixed point based on the "iteration+widening" strategy block by block. During the analysis, we translate the interval representation of the pre-state to an SMT formula at the entry of the block, and translate the post-state in SMT formula back to interval representation.

6.2 Constrained Random Fuzzing with TaichiPoly

Constrained random verification [Kitchen and Kuehlmann 2007; Naveh and Metodi [n. d.]; Wu and Huang 2013] is widely used for validating hardware designs. The verification engineers specify the constraint required by the hardware and then generate multiple random inputs satisfying the constraint using a stimulus generator. These inputs are used to drive the design under test, in an attempt to cover the design space and trigger faults.

Recently, the idea has been lifted to dynamic program analysis, yielding the notion of *constrained random fuzzing* [Dutra et al. 2018; Huang et al. 2020]. The basic observation is that many feasible program paths share the same path prefix. Constrained random fuzzing attempts to generate multiple models for the path constraint of a selected path prefix. These models/inputs can then examine multiple program paths that share the same prefix, without additional invocations of an expensive constraint solver for each path.

We have adapted a previous approach that utilizes polyhedral abstraction in constrained random fuzzing [Huang et al. 2020]. First, we compute the symbolic polyhedral abstraction of a satisfiable path constraint. By this means, we can convert the problem of generating models of the constraint into the problem of sampling integral points in the polyhedron [Chen et al. 2018; Kannan and Narayanan 2009, 2012].



(a) The template polyhedron computed by [Huang et al. 2020].

(b) The polyhedron computed by TaichiPoly.

Fig. 5. A formula $\varphi \equiv (x = 0 \wedge y = 0) \vee (x = 2 \wedge y = 3) \vee (x = 3 \wedge (y = 1 \vee y = 2))$.

Example 6.1. Consider the formula φ in Figure 5, and let it be the path condition of a selected path prefix. The red dots represent all feasible values satisfying φ, whereas the black crosses represent the infeasible ones. The abstractions of φ are the orange regions bounded by the lines representing multiple linear inequalities. Figure 5a depicts the template polyhedral abstraction computed by Huang et al. [2020]'s approach, where the templates consist of (1) all variables in $vars(\varphi)$ and (2) linear expressions over $vars(\varphi)$ that occur in the formula. In comparison, Figure 5b shows the polyhedron computed by TaichiPoly.

As introduced above, constrained random fuzzing aims to generate models of the original formula φ. Here, if we perform a uniform sampling of integral points in the two polyhedrons, the success rate (the sampled points satisfy φ) would be $4/16 = 25\%$ and $4/7 = 57\%$, respectively. Clearly, this examples shows that the precision advantage of TaichiPoly (Figure 5b) has the potential for scalability benefits.

We implement the client on top of Qsym [Yun et al. 2018], a hybrid fuzzing framework that combines symbolic execution and a mutational fuzzer (AFL [afl 2014]). The system uses AFL to test most easy-to-cover branches quickly. For a hard-to-cover branch, we collect the path constraint using Qsym's symbolic execution engine, compute its polyhedral abstraction via TaichiPoly, and apply a Markov Chain Monte Carlo sampling algorithm [Kannan and Narayanan 2009] to uniformly sample points over the polyhedron. Note that, for this client, we enforce the absence of overflows in the formulas to be abstracted, using encoding schema presented in § 5. Thus, we end up with an under-approximation that excludes some possible behaviors of the program from consideration.

7 EVALUATION

This section evaluates the performance of TaichiInt and TaichiPoly, and their effectiveness when used in the two clients, memory corruption analysis and constrained random fuzzing.

Benchmarks. For the memory corruption analysis, we use the DARPA's Cyber Grand Challenge (CGC) dataset [Song and Alves-Foss 2016], which consists of 131 different binaries. The sizes of those binaries range from 83 KB to 18 MB. These programs contain 28 heap overflow bugs, 24 stack overflow bugs, 16 null pointer dereference bugs, 13 integer overflow bugs, and 8 use-after-free bugs. The dataset is designed to exhibit diversified and common code patterns, and has been widely used as a test suite for automated vulnerability detection and exploitation systems [Bao et al. 2017; Poeplau and Francillon 2020; Stephens et al. 2016]. We configure our tool to detect stack overflow and use-after-free bugs. We impose a 60 second time limit for analyzing any individual function. If

Table 2. Real-world benchmark programs for constrained-random fuzzing.

Project	KLoC	Binary	Version	Input format
libjpeg	289	libjpg	commit-ec5adb	JPG
jhead	5.4	jhead	3.03	JPG
bento4	180.0	MP42aac	commit-cbebcc	MP4
tcpdump	97.2	tcpdump	commit-b5046f	PCAP
libtiff	119.3	Tiffops	4.0.10	TIFF
binutils	764.7	nm-new	2.33	ELF
binutils	764.7	readelf	2.33	ELF
binutils	764.7	objdump	2.33	ELF
openssl	436.4	asn1parse	commit-e8d01	ASN
coreutils	230.7	uniq	LAVA-M	TXT

the analysis of a function times out, it is regarded as a function that returns a nondeterministic value and has non-deterministic side-effects on variables passed by reference.

For the constrained-random-fuzzing client, we have evaluated on ten binaries from eight projects, including `libjpg`, `jhead`, `MP42aac`, `tcpdump`, `Tiffops`, `nm-new`, `readelf`, `objdump`, `asn1parse`, and `uniq`. Table 2 shows the size, version, and input format of the projects and binaries. These binaries have diverse functionalities and complexity, most of which have been widely evaluated by existing fuzz testing tools [Chen and Chen 2018; Dolan-Gavitt et al. 2016; Yun et al. 2018]. We set a twenty-four-hour time budget for testing each binary, and export the path constraints used for computing the polyhedral abstractions.

In total, we collect 50,299 interval abstraction queries and 7,634 polyhedral abstraction queries, respectively. The queries are exposed in SMT-LIB2 format, allowing us to compare with other off-the-shelf tools. Note that the SMT formulas generated by Angr and Qsym are translated from VEX IR and x86 assembly, respectively.

Platform. We conduct the experiments on an 80-core 2.20 GHz CPU with 256 GB of memory running Ubuntu 16.04. Though the processor is multi-core, the executables themselves are single-threaded. We repeat each experiment ten times and report the average results.

7.1 Interval Abstraction

Better Performance than Existing Approaches to Symbolic Interval Abstraction. The state-of-the-art approach to symbolic interval abstraction is to reduce it to OMT solving problems. In this experiment, we compare TaichiInt with three groups of techniques for solving OMT instances in bit-vector theory. In what follows, we briefly illustrate the baseline approaches using the optimization problem "max x $s.t.$ φ", where φ is a quantifier-free bit-vector formula, and x encodes an n-bit bit vector. We assume x is an unsigned integer to ease the explanation of the baseline approaches.

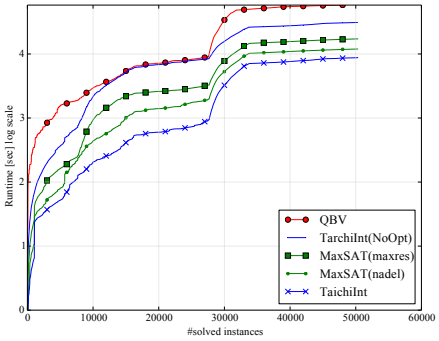
(1) *Existing OMT solvers.* Most state-of-the-art solvers reduce "max x $s.t.$ φ" to a weighted MaxSAT problem [Bjørner et al. 2015]. The above optimization problem is encoded as

$$\begin{cases} \text{Hard constraints} & \text{translate } \varphi \text{ to a SAT formula} \\ \text{Soft constraints} & \begin{cases} (t_0 \text{ weight } 1) \wedge \\ (t_1 \text{ weight } 2) \wedge \\ \quad \ldots \quad \wedge \\ (t_{n-1} \text{ weight } 2^{n-1}) \end{cases} \end{cases}$$

within two steps. First, the formula φ is translated to a hard Boolean formula via bit-blasting [Barrett and Tinelli 2018], where x is represented as a sequence of Boolean variables

Algorithm	Time(s)	# Unsolved
Qbv	39,038	346
MaxSAT(maxres)	5,882	67
MaxSAT(nadel)	4,704	38
TaichiInt(NoOpt)	10,471	80
TaichiInt	2,212	26

(a) Total solving time and unsolved queries.



(b) Cactus plot for all solved instances.

Fig. 6. Results of running each solver for the interval abstraction queries.

$[t_{n-1}, \ldots, t_0]$. Second, for each t_i, soft weighted unit clause (t_i) of the weight 2^i is added. Then, the objective is to maximize the value of $t_0 * 1 + t_1 * 2 + \cdots + t_{n-1} * 2^{n-1}$ via an MaxSAT solving algorithm. In the experiment, we evaluated two MaxSAT solving algorithms implemented in νZ [Bjørner et al. 2015]: core-guided maximal resolution (MaxSAT(maxres)) [Narodytska and Bacchus 2014], and Nadel and Ryvchin [2016]'s algorithm (MaxSAT(nadel)). We use the boxed multi-objective optimization mode of νZ.

(2) *Quantified SMT solving*. We can compute the optimal values of a variable via encoding and solving quantified formulas [Kong et al. 2018]. For example, the optimization problem "max x $s.t.$ φ" can be encoded as

$$\Psi \equiv \varphi \wedge (\forall x'.\varphi[x'/x] \rightarrow x \geq x') \ ,$$

which states that for any other variable x' such that $\varphi[x'/x]$ is satisfiable, we have $x \geq x'$. If M is a model of Ψ, then $M(x)$ is the maximum value of x. To solve the translated formulas, we use Z3's decision procedure for quantified bit-vector constraints (denoted "Qbv") [Wintersteiger et al. 2013].

(3) TaichiInt(NoOpt). It uses an SMT-based binary search as TaichiInt, except that it does not apply the sound interval analysis and finds solutions for multiple objectives independently, without reusing models amongst the objectives.

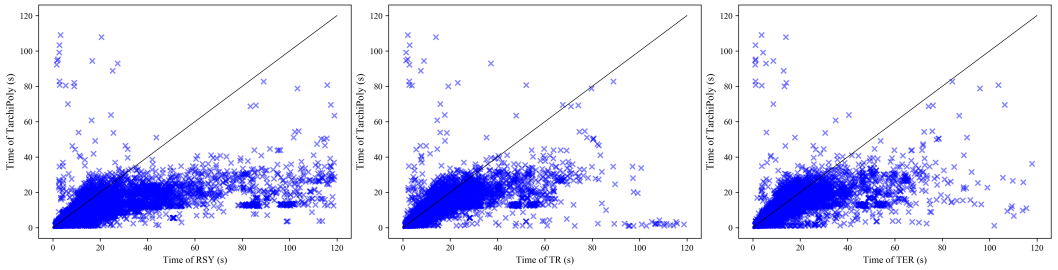
Figure 6a and Figure 6b summarize the results of running the five algorithms for interval abstraction queries, with a timeout of 30 seconds per query. On average, TaichiInt obtains 2.1× to 17.6× speedups over MaxSAT(maxres), MaxSAT(nadel), and Qbv. Besides, TaichiInt can solve 12, 44, and 320 more interval queries than MaxSAT(maxres), MaxSAT(nadel), and Qbv, respectively.

Compared with TaichiInt(NoOpt), TaichiInt solves 114 more queries and is on average 4.7× faster. The major reason is that optimizing multiple objectives simultaneously (as TaichiInt does) ensures that all objectives benefit from the sampled models and potentially avoids repeating expensive SMT calls. Besides, we also notice that the sound interval analysis in TaichiInt can often reduce the number of SMT queries by 8% to 21%. We anticipate that the search space could be better narrowed down by a sound and more precise interval analysis for bit-vectors. However, exploring this direction is non-trivial, as much of the literature on interval analysis uses unbounded integers [Gange et al. 2015].

Better Precision for Memory Corruption Analysis. We examine the precision of TaichiInt by integrating it in a tool for detecting memory corruption bugs (§ 6.1). The tool uses a conventional interval analysis that computes sound but not necessarily best intervals. We then extend the tool

Table 3. Results of detecting vulnerabilities in the CGC dataset.

Tool	# Reports	# TP	FP Rate
Angr(ConvInt)	124	25	79.8%
Angr(TaichiInt)	58	25	56.9%



(a) TaichiPoly vs. Rsy (b) TaichiPoly vs. Tr (c) TaichiPoly vs.Ter

Fig. 7. Performance comparison for symbolic polyhedral abstraction.

Table 4. Results of running each solver for polyhedral abstraction on 7,634 instances.

Algorithm	Time(s)	# Unsolved
Rsy	246,418	429
Tr	177,458	116
Ter	167,817	215
TaichiPoly	68,499	65

with TaichiInt to measure the number of false positives TaichiInt can reduce. Table 3 summarizes the analysis results on the CGC dataset. We only report the results for programs that cannot be proven safe by Angr(ConvInt). When using conventional interval analysis, Angr(ConvInt) is able to identify 124 vulnerabilities while producing 99 false positives, resulting in a false-positive rate of 79.8%. Armed with TaichiInt, the false-positive rate of Angr(TaichiInt) is reduced by 22.9%, with 66 false positives removed. [10]

7.2 Polyhedra Abstraction

Better Performance than Existing Approaches to Symbolic Polyhedral Abstraction. Recall that we cannot reduce symbolic polyhedral abstraction to OMT solving, because the number and coefficients of linear inequalities are unknown. Thus, we compare TaichiPoly against the following algorithms in the literature of symbolic abstraction.

(1) Reps et al. [2004]'s algorithm that starts from lattice $\bot$ (denoted "Rsy").
(2) Thakur and Reps [2012]'s algorithm that starts from lattice $\top$ (denoted "Tr").
(3) Thakur et al. [2012]'s Bilateral algorithm that starts from both lattice $\bot$ and lattice $\top$ (denoted "Ter").

Note that Tr and Ter were previously used for computing the polyhedral abstractions of QF_LRA formulas. We have adapted the three baselines to handle QF_BV, and implemented them using Z3.

Note that, to improve the empirical effectiveness of the client, we enforce the absence of overflows in the formulas to be abstracted, by using the encoding schema discussed in § 5.

[10]Note that, both Angr(ConvInt) and Angr(TaichiInt) miss some stack overflow and use-after-free bugs, which is mainly caused by some unmodeled instructions.

Table 4 and Figure 7 show the results of running the algorithms with a timeout of 120 seconds per query. Table 4 compares the total runtime and the number of unsolved queries. Figure 7 is the scatter plot for the solved queries, where axes correspond to the CPU time (measured in seconds) taken by TaichiPoly (y-axis) and a baseline technique (x-axis). Each point on the figure represents a query. The points below the diagonal represent problems where TaichiPoly is faster.

The results show that TaichiPoly outperforms the baselines on our set of benchmarks in most cases. The average speed up of TaichiPoly vs. Rsy, Tr, and Ter. are 3.6×, 2.6×, 2.4×, respectively. Besides, TaichiPoly can solve 364, 51, and 150 more queries than Rsy, Tr, and Ter, respectively.

Besides, in our experiments, we also observe that over 83% of the polyhedral abstraction queries can be solved by TaichiPoly within 15 seconds. Our findings show that the overly pessimistic view of symbolic abstraction could lead researchers to underestimate its potential applications. Computing the best polyhedral abstractions, the conventionally most expressive domain, can be efficient for queries from large and realistic programs.

Better Precision for Constrained Random Fuzzing. In this study, we apply TaichiPoly to the client of constrained-random fuzzing (§ 6.2). Given a path constraint, the goal is to generate a set of models satisfying the constraint. In this experiment, we compared two algorithms:

(1) Pangolin(Templates) [Huang et al. 2020] computes the symbolic abstraction of the template polyhedral domain. First, it extracts input variables and their linear expressions in the formula as the templates, and uses an OMT solver to compute the lower and upper bounds of the templates (as in § 2.2). Then, it generates candidate models of the formula by sampling integral points in the polyhedron.
(2) Pangolin(TaichiPoly) uses the same sampling algorithm as Pangolin(Templates), but computes the polyhedral abstraction via TaichiPoly.

We exclude Rsy, Tr, and Ter in this study, because they compute the same polyhedron as TaichiPoly, and the difference lies in the scalability. Besides, we tried our best to run Elina [Singh et al. 2017a], the state-of-the-art, conventional polyhedral analysis. However, it is far from trivial to offer an apples-to-apples comparison. First, TaichiPoly over-approximates formulas translated from the assembly, but Elina does not have front ends supporting SMTLIB2 or assembly. Second, Elina requires a third-party pointer analysis to reason about aliasing, which is a source of imprecision [Wei et al. 2018]. While in the assembly-level symbolic execution, such information has been precisely and implicitly encoded in the formulas.

We compare the runtime performance for all the benchmarks whose polyhedral abstraction can be computed within 15 seconds. For each benchmark, we generate between 5,000 and 50,000 samples (depending on the size of the benchmark) and compute the average time taken to generate a valid (satisfying) and unique sample. Figure 8 shows the average results for queries from each project. We observe that Pangolin(TaichiPoly) achieves a higher sampling speed than Pangolin(Templates) on most cases. This is mainly because that, compared with Pangolin(Templates), the polyhedral abstraction produced by TaichiPoly is often more precise. As a result, the sampling procedure can be more effective, because it operators over a smaller polyhedral region.

7.3 Discussions

Summary of Experiments. The experiments compare our algorithms with existing techniques in the OMT solving and symbolic abstraction literature, which demonstrate the power of our approach. We also evaluate two program analysis clients, including (1) the flow-sensitive, context-sensitive interprocedural memory corruption analysis, where the abstract transformers of a selected set of functions are created block-wise via symbolic interval abstraction; and (2) the interprocedurally path-sensitive constrained random fuzzing, where we compute the symbolic polyhedral abstractions

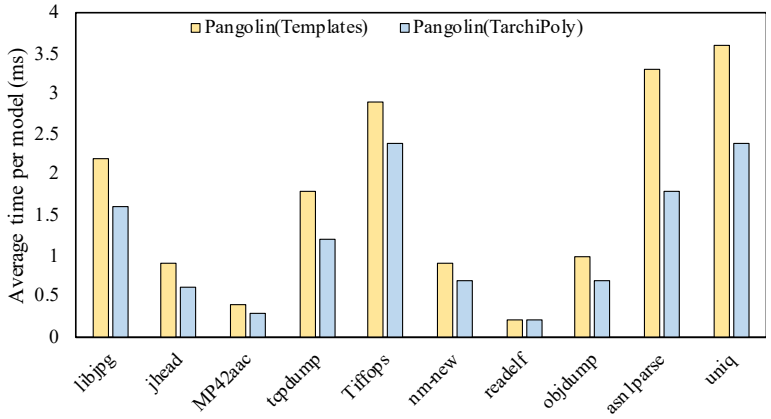


Fig. 8. Comparison of sampling speed. The y-aix shows the average time (in millisecond) of sampling one valid model.

for the path conditions of selected path prefixes. Specifically, the sizes of the real-world projects for constrained random fuzzing range from 5.4K to 764.7K lines of code. The results highlight the effectiveness of our algorithms for realistic and large-scale programs.

Convergence in Lazy All-SMT. Many algorithms in automated reasoning work in the style of the lazy All-SMT loop [John and Chakraborty 2011; Lahiri et al. 2006; Monniaux 2010; Sullivan et al. 2019]. These algorithms iteratively enumerate the models of a formula φ, which are used to "generalize" some abstraction; the iteration terminates until the abstraction has encompassed the formula's solution space. Three factors affect the scalability: the model returned by the decision procedure, the method for the generalization, and the performance of the decision procedure. Many of the past efforts target the latter two factors. For example, Thakur and Reps [2012] optimize the generalization in symbolic abstraction by computing the "abstract consequences" of an atomic formula. Lahiri et al. [2006]'s predicate abstraction algorithm speeds up SMT solving by guiding the conflict-driven back-jumping inside SMT solvers.

The tenet of our work is to better reuse models and find "better" models for speeding up the convergence. An interesting merit is the use of interval abstraction for accelerating polyhedral abstraction, which opens up a new connection between OMT solving and symbolic abstraction. Note that Algorithm 6 and Algorithm 7 do not attempt to refine the intervals (in the sense of abstraction refinement). Instead, they leverage the specific models under which the variables have the extremal values.

Sampling and Optimization. Our work opens up a dual-use of solution sampling and constrained optimization. On the one hand, our algorithm attempts to sample "better" models of a formula for accelerating symbolic abstraction (a form of optimization). On the other hand, we demonstrate that a formula's symbolic abstractions can be used to sample its solutions. We do not claim that Pangolin(TaichiPoly) is superior than existing general-purpose uniform samplers such as Unigen3 [Meel and Akshay 2020]. The benefits of Pangolin(TaichiPoly) are correlated with the size of the formula, the number of the formula's variables, the model count of the formula, to name a few. For example, MCMC methods can suffer from the curse of dimensionality [Chernozhukov and Hong 2003; Ermon et al. 2013], which means the possibility of sampling inside a certain space

in the target object decreases very quickly while the dimension increases. However, we believe that our work represents an interesting point in the connections between sampling and optimization.

Applicability of Symbolic Abstraction. Symbolic abstraction is an instance of a fundamental approximation problem: given a formula φ in a logic $\mathcal{L}$ and a less expressive logic $\mathcal{L}'$, find the strongest consequence of φ that is expressible in $\mathcal{L}'$ [Reps and Thakur 2016]. For instance, for the interval domain, the logic $\mathcal{L}'$ can be regarded as a logic fragment of conjunctions of single-variable inequalities. In the context of abstract interpretation, beyond computing the best abstract transformer, the symbolic abstraction also gives a way to automate other operations, such as (1) performing semantic reduction [Cousot and Cousot 1979; Kincaid et al. 2017], (2) performing reduced-product calculations [Thakur et al. 2015], and (3) converting an abstract value from one abstract domain to another (e.g., when crossing analysis boundaries), etc. For example, let ψ be a constraint representation of the polyhedral domain. We can convert ψ to an element in the octagon domain, by converting ψ to a first-order formula and performing symbolic abstraction to ψ.

8 RELATED WORK

We discuss closely-related work in two groups: symbolic abstraction and automated reasoning.

8.1 Symbolic Abstraction

Reps et al. [2004] introduce the problem of symbolic abstraction, which computes the best abstraction of a formula in a given abstract domain. The problem has been undertaken for finite-height domains [Reps et al. 2004], template linear domains [Brauer and King 2010; Monniaux 2009], the wedge domains [Kincaid et al. 2017], as well as the polyhedral domains [Thakur and Reps 2012]. Symbolic abstraction has found many applications, such as shape analysis [Reps et al. 2004; Yorsh et al. 2004], program verification [Jiang et al. 2017; Li et al. 2014], control flow recovery [Barrett and King 2010], and compiler optimization [Ritter 2015]. In what follows, we focus on algorithms for symbolic abstraction.

Algorithmic Framework. The RSY algorithm [Reps et al. 2004] is a parametric framework that applies to different domains. To abstract a formula φ, it iteratively calls a decision procedure to sample a model of φ, and generalizes the current abstraction using the model, until the abstraction encompasses all models of φ. A limitation of RSY is that it is not resilient to timeouts. If running out of the time budget, it must return $\top$ to be sound as it starts from lattice $\bot$. In comparison, Thakur and Reps [2012]'s algorithm starts from lattice $\top$. Thakur et al. [2012]'s Bilateral algorithm starts from both lattice $\bot$ and lattice $\top$ (denoted "TER"). Both TR and TER could return a nontrivial (non $\top$) value in case of a timeout [Thakur et al. 2012; Thakur and Reps 2012]. TAICHIPOLY builds on the RSY framework, and centers around a critical but long-neglected design bottleneck in symbolic abstraction, the way for sampling the models. Specifically, our approach attempts to find "better" models to speed up the convergence.

Interval Domain. Regehr and Reid [2004] present a method that constructs the best abstract transformers for machine instructions, for the interval and bitwise abstract domains. Their method does not call a SAT or SMT solver but, instead, uses the physical processor as a black box. To compute the abstract post-state for an abstract value a, the approach recursively divides a until an abstract value, whose concretization is a singleton set, is obtained. Barrett and King [2010] develop a method of generating interval and set abstractions for bit-vectors that are bit-blasted to Boolean formulas. For interval analysis, they separately compute the minimum and maximum value of the range for an n-bit bit-vector using $2n$ calls to an SAT solver, with each SAT query determining a single bit of the output. Barrett and King [2010]'s algorithms only consider the abstraction of one variable, whereas our interval abstraction algorithm targets multiple variables.

Polyhedral Domain. While several efforts have been made for the symbolic abstractions of template linear domains, little work has been done for polyhedral domains. Thakur et al. [2012]'s algorithm and Thakur and Reps [2012]'s algorithm have only been applied to the symbolic polyhedral abstraction of QF_LRA formulas. Our work provides the first empirical comparison of the algorithms for QF_BV. When formulating this work, we noticed that Algorithm 6 is similar to an unpublished algorithm by Jörg Brauer.[11] Our polyhedral analysis differs in two aspects. First, our solution Algorithm 7 interleaves the RSY algorithm and Algorithm 6, aiming to balance the cost and quality of the sampled models. Second, our algorithms simultaneously compute the symbolic interval abstractions of different variables, while Brauer's algorithm computes such information variable by variable.

In the realm of conventional polyhedral analysis, Elina's online decompositions improve the performance of the polyhedral domain by a large margin [Singh et al. 2017a,c]. It is further optimized with learning-based heuristics [He et al. 2020; Singh et al. 2018] that trade precision for scalability. Elina has mainly been used for whole-program analysis, while we apply TaichiPoly to compute the polyhedral abstraction for specific path constraints. We found that TaichiPoly can be extremely expensive when used for an exhaustive analysis of large-scale programs.

Apart from the abstract interpretation community, there are alternatives for deriving sound polyhedral approximations of a formula. For example, Gröbner basis algorithms allow for deducing linear inequalities from a non-linear formula over fields, such as rationals and complex numbers [Becker et al. 1993]. Recently, a variant of Buchberger's algorithm has been reported that is applicable to modulo integers with respect to arbitrary moduli [Brickenstein et al. 2009]. Besides, one could apply machine learning techniques such as SVM to learn linear over-approximations of a (non-linear) formula, by utilizing both satisfying and falsifying assignments of the formula [Dathathri et al. 2017]. However, both the Gröbner basis-based approach [Brickenstein et al. 2009] and the learning-based approach [Dathathri et al. 2017] have no guarantees on the precision of the derived abstractions.

8.2 Automated Reasoning

Symbolic abstraction is also closely related to several problems in automated reasoning, including optimization modulo theories, quantified constraint satisfaction, and quantifier elimination.

Optimization Modulo Theories. OMT is an extension of SMT that allows for finding models optimizing given objectives [Fazekas et al. 2018; Li et al. 2014; Nieuwenhuis and Oliveras 2006; Sebastiani and Tomasi 2015b; Sebastiani and Trentin 2015a,b]. Previous work [Jiang et al. 2017; Li et al. 2014] shows that symbolic abstraction of template linear domains such as interval [Cousot and Cousot 1977] and octagon [Miné 2006] can be reduced to solving boxed OMT problems. A closely related work is Symba [Li et al. 2014], which also simultaneously optimizes multiple objectives and reuses information between them to speed up the analysis. There are three main differences. First, Symba targets unbounded and linear arithmetic, while we focus on bit-vectors. Second, Symba is a linear-search-style algorithm, while our procedure for interval abstraction is binary-search-style. Third, Symba is only applicable to the template linear domains, while we further target the polyhedral domain.

Most state-of-the-art OMT solvers such as νZ [Bjørner et al. 2015] and OpTiMathSAT [Sebastiani and Tomasi 2015a] reduce the problem of bit-vector optimization to weighted MaxSAT solving [Bjørner et al. 2015; Sebastiani and Tomasi 2015a]. Nadel and Ryvchin [2016]'s approach first transforms a bit-vector formula to an SAT formula and then performs a binary search exploration over the bits of the objective function, using a sequence of incremental calls to the underlying

[11] Personal communication.

SAT solver. Their approach is essentially a variant of Algorithm 1 but does not consider boxed multi-objectives optimization problems.

Quantified Constraint Satisfaction. Many OMT problems can be addressed by solving quantified constraint satisfaction problems. For instance, previous work has shown that a decision procedure for solving quantified formulas can effectively solve the OMT problem specific for non-linear real arithmetic [Kong et al. 2018]. This paper offers the first empirical evidence on the performance of quantified bit-vector solving for symbolic interval abstraction. To solve quantified bit-vector formulas, Z3 combines model-based quantifier instantiation [De Moura and Bjørner 2008] and a model finding procedure based on templates [Wintersteiger et al. 2013]. Q3B [Jonás and Strejcek 2016] translates the formulas to the Binary Decision Diagram, and handles non-linear operations by approximations. Recently, the Boolector and CVC4 developers introduced the CEGIS (counter-example guided inductive synthesis) methodology to solve quantified formulas [Niemetz et al. 2018, 2021; Preiner et al. 2017]. Despite the progress, we find that state-of-the-art SMT solvers still often run into difficulties when solving quantified formulas translated from large OMT instances in the bit-vector theory.

Quantifier Elimination. Symbolic abstraction also has a connection to the problem quantifier elimination [Arnon 1988; Loos and Weispfenning 1993] or "forgetting" [Lin 2001; Lin and Reiter 1994], which can compute the strongest consequence of a formula that mentions only a subset of its variables. Gulwani and Musuvathi [2008] define the "cover problem" that essentially approximates existential quantifier elimination for the combined theory of uninterpreted functions and linear arithmetic. Monniaux [2009] introduced a quantifier-elimination-based approach to computing optimal abstract transformers of template linear domains, such as intervals, octagon, and template polyhedron. Monniaux [2009]'s algorithm targets linear real arithmetic. To date, there has been relatively little progress on quantifier elimination for the bit-vector theory [Backeman et al. 2018]. Translation from bit-vectors to unbounded arithmetic can result in complicated constraints that are hard to reason about, and bit-blasting to propositional logic leads to an exponential increase in the formula size. The algorithms by John and Chakraborty [2011, 2013] only support the linear modulo arithmetic. Model-based projection [Komuravelli et al. 2016] can compute under-approximations of quantifier elimination, which does not guarantee precision.

9 CONCLUSION

The development of abstract domains has never been easy, which requires significant expertise, careful tuning of the abstract transformers, as well as code optimizations. Symbolic abstraction has the significant potential to change this landscape. It allows for the automatic synthesis of abstract transformers for a block of code, which could be more precise than the usual composition of the individual operations' abstractions, and less tedious to design and implement transfer functions. This paper provides strong evidence that symbolic abstraction of numeric domains can be made efficient and practical for large and realistic programs. In a virtuous cycle, the development and widespread use of symbolic abstraction algorithms would likely uncover additional client analyses that benefit from the added precision.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd for valuable feedback on earlier drafts of this paper, which helped improve its presentation. We also appreciate Dr. Jörg Brauer and Dr. Mianlai Zhou for insightful discussions. The authors are supported by the RGC16206517 and ITS/440/18FP grants from the Hong Kong Research Grant Council, Ant Research Program, and the donations from Microsoft and Huawei. Qingkai Shi is the corresponding author.

REFERENCES

2014. AFL: american fuzzy lop. http://lcamtuf.coredump.cx/afl/. Accessed: 2014.

Bijan Alizadeh and Masahiro Fujita. 2009. Modular arithmetic decision procedure with auto-correction mechanism. In *IEEE International High Level Design Validation and Test Workshop, HLDVT 2009, San Francisco, CA, USA, 4-6 November 2009*. IEEE Computer Society, 138–145.

Dennis S. Arnon. 1988. A Bibliography of Quantifier Elimination for Real Closed Fields. *J. Symb. Comput.* 5, 1/2 (1988).

Benjamin Assarf, Ewgenij Gawrilow, Katrin Herr, Michael Joswig, Benjamin Lorenz, Andreas Paffenholz, and Thomas Rehn. 2017. Computing convex hulls and counting integer points with polymake. *Math. Program. Comput.* 9, 1 (2017), 1–38.

Peter Backeman, Philipp Rümmer, and Aleksandar Zeljic. 2018. Bit-Vector Interpolation and Quantifier Elimination by Lazy Reduction. In *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, Nikolaj Bjørner and Arie Gurfinkel (Eds.). IEEE, 1–10.

Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, and David Brumley. 2017. Your Exploit is Mine: Automatic Shellcode Transplant for Remote Exploits. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 824–839.

Clark W. Barrett and Cesare Tinelli. 2018. Satisfiability Modulo Theories. In *Handbook of Model Checking*, Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (Eds.). Springer, 305–343.

Edd Barrett and Andy King. 2010. Range and Set Abstraction using SAT. *Electron. Notes Theor. Comput. Sci.* 267, 1 (2010).

Thomas Becker, Volker Weispfenning, and Heinz Kredel. 1993. *Gröbner bases - a computational approach to commutative algebra*. Graduate texts in mathematics, Vol. 141. Springer.

Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M. Erkan Keremoglu, and Roberto Sebastiani. 2009. Software model checking via large-block encoding. In *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*. IEEE, 25–32.

Fabrizio Biondi, Michael A. Enescu, Annelie Heuser, Axel Legay, Kuldeep S. Meel, and Jean Quilbeuf. 2018. Scalable Approximation of Quantitative Information Flow in Programs. In *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10747)*, Isil Dillig and Jens Palsberg (Eds.). Springer, 71–93.

Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. 2015. νZ-an optimizing SMT solver. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9035*. Springer-Verlag, Berlin, Heidelberg, 194–199.

Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome 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* (San Diego, California, USA) *(PLDI '03)*. ACM, New York, NY, USA, 196–207.

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-Verlag, Berlin, Heidelberg, 167–183.

Michael Brickenstein, Alexander Dreyer, Gert-Martin Greuel, Markus Wedler, and Oliver Wienand. 2009. New developments in the theory of Gröbner bases and applications to formal verification. *Journal of Pure and Applied Algebra* 213, 8 (2009).

Stefan Bygde, Björn Lisper, and Niklas Holsti. 2012. Fully Bounded Polyhedral Analysis of Integers with Wrapping. *Electron. Notes Theor. Comput. Sci.* 288 (2012), 3–13.

P. Chen and H. Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*, Vol. 00. 711–725.

Yuansi Chen, Raaz Dwivedi, Martin J. Wainwright, and Bin Yu. 2018. Fast MCMC Sampling Algorithms on Polytopes. *J. Mach. Learn. Res.* 19 (2018), 55:1–55:86.

Victor Chernozhukov and Han Hong. 2003. An MCMC approach to classical estimation. *Journal of Econometrics* 115, 2 (2003), 293–346.

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). Association for Computing Machinery, New York, NY, USA, 77–94.

Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (San Antonio, Texas) *(POPL '79)*. Association for Computing Machinery, New York, NY, USA, 269–282.

Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski (Eds.). ACM Press, 84–96.

Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Rosu. 2019. A complete formal semantics of x86-64 user-level instruction set architecture. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley

and Kathleen Fisher (Eds.). ACM, 1133–1148.

Sumanth Dathathri, Nikos Aréchiga, Sicun Gao, and Richard M. Murray. 2017. Learning-Based Abstractions for Nonlinear Constraint Solving. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, Carles Sierra (Ed.). ijcai.org, 592–599.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. 2016. LAVA: Large-Scale Automated Vulnerability Addition. In *2016 IEEE Symposium on Security and Privacy (SP)*. 110–121.

Rafael Dutra, Kevin Laeufer, Jonathan Bachrach, and Koushik Sen. 2018. Efficient Sampling of SAT Solutions for Testing. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 549–559.

Martin E. Dyer and Alan M. Frieze. 1988. On the Complexity of Computing the Volume of a Polyhedron. *SIAM J. Comput.* 17, 5 (1988), 967–974.

Stefano Ermon, Carla P. Gomes, Ashish Sabharwal, and Bart Selman. 2013. Taming the Curse of Dimensionality: Discrete Integration by Hashing and Optimization. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013 (JMLR Workshop and Conference Proceedings, Vol. 28)*. JMLR.org, 334–342.

Katalin Fazekas, Fahiem Bacchus, and Armin Biere. 2018. Implicit Hitting Set Algorithms for Maximum Satisfiability Modulo Theories. In *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10900)*, Didier Galmiche, Stephan Schulz, and Roberto Sebastiani (Eds.). Springer, 134–151.

Vijay Ganesh and David L Dill. 2007. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification* (Berlin, Germany) *(CAV'07)*. Springer-Verlag, Berlin, Heidelberg, 519–531.

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 Trans. Program. Lang. Syst.* 37, 1, Article 1 (Jan. 2015), 1:1–1:35 pages.

Denis Gopan and Thomas Reps. 2007. Low-level Library Analysis and Summarization. In *Proceedings of the 19th International Conference on Computer Aided Verification* (Berlin, Germany) *(CAV'07)*. Springer-Verlag, Berlin, Heidelberg, 68–81.

Sumit Gulwani and Madan Musuvathi. 2008. Cover Algorithms and Their Combination. In *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4960)*, Sophia Drossopoulou (Ed.). Springer, 193–207.

Jingxuan He, Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2020. Learning fast and precise numerical analysis. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 1112–1127.

Julien Henry, Mihail Asavoae, David Monniaux, and Claire Maiza. 2014. How to compute worst-case execution time by optimization modulo theory and a clever encoding of program semantics. In *SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2014, LCTES '14, Edinburgh, United Kingdom - June 12 - 13, 2014*, Youtao Zhang and Prasad Kulkarni (Eds.). ACM, 43–52.

Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. 2020. Pangolin: Incremental Hybrid Fuzzing with Polyhedral Path Abstraction. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1613–1627.

Jiahong Jiang, Liqian Chen, Xueguang Wu, and Ji Wang. 2017. Block-Wise Abstract Interpretation by Combining Abstract Domains with SMT. In *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10145)*, Ahmed Bouajjani and David Monniaux (Eds.). Springer, 310–329.

Ajith K John and Supratik Chakraborty. 2011. A quantifier elimination algorithm for linear modular equations and disequations. In *Proceedings of the 23rd International Conference on Computer Aided Verification* (Snowbird, UT) *(CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 486–503.

Ajith K John and Supratik Chakraborty. 2013. Extending quantifier elimination to linear inequalities on bit-vectors. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Rome, Italy) *(TACAS'13)*. Springer-Verlag, Berlin, Heidelberg, 78–92.

Martin Jonás and Jan Strejcek. 2016. Solving Quantified Bit-Vector Formulas Using Binary Decision Diagrams. In *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9710)*, Nadia Creignou and Daniel Le Berre (Eds.). Springer, 267–283.

Ravi Kannan and Hariharan Narayanan. 2009. Random Walks on Polytopes and an Affine Interior Point Method for Linear Programming. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing* (Bethesda, MD, USA)

(STOC '09). ACM, New York, NY, USA, 561–570.

Ravindran Kannan and Hariharan Narayanan. 2012. Random Walks on Polytopes and an Affine Interior Point Method for Linear Programming. *Math. Oper. Res.* 37, 1 (2012), 1–20.

Zachary Kincaid, John Cyphert, Jason Breck, and Thomas Reps. 2017. Non-Linear Reasoning for Invariant Synthesis. *Proc. ACM Program. Lang.* 2, POPL, Article 54 (Dec. 2017), 33 pages.

Nathan Kitchen and Andreas Kuehlmann. 2007. Stimulus generation for constrained random simulation. In *2007 International Conference on Computer-Aided Design, ICCAD 2007, San Jose, CA, USA, November 5-8, 2007*, Georges G. E. Gielen (Ed.). IEEE Computer Society, 258–265.

Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. 2012. Constraints as Control. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 151–164.

Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2016. SMT-based model checking for recursive programs. *Formal Methods Syst. Des.* 48, 3 (2016), 175–205.

Soonho Kong, Armando Solar-Lezama, and Sicun Gao. 2018. Delta-Decision Procedures for Exists-Forall Problems over the Reals. 10982 (2018), 219–235.

Shuvendu K Lahiri, Robert Nieuwenhuis, and Albert Oliveras. 2006. SMT techniques for fast predicate abstraction. In *Proceedings of the 18th International Conference on Computer Aided Verification* (Seattle, WA) *(CAV'06)*. Springer-Verlag, Berlin, Heidelberg, 424–437.

Yi Li, Aws Albarghouthi, Zachary Kincaid, Arie Gurfinkel, and Marsha Chechik. 2014. Symbolic optimization with SMT solvers. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. ACM, New York, NY, USA, 607–618.

Junghee Lim and Thomas Reps. 2013. TSL: A System for Generating Abstract Interpreters and Its Application to Machine-Code Analysis. *ACM Trans. Program. Lang. Syst.* 35, 1, Article 4 (April 2013), 59 pages.

Fangzhen Lin. 2001. On strongest necessary and weakest sufficient conditions. *Artif. Intell.* 128, 1-2 (2001), 143–159.

Fangzhen Lin and Ray Reiter. 1994. Forget it. In *Working Notes of AAAI Fall Symposium on Relevance*. 154–159.

Björn Lisper. 2003. Fully Automatic, Parametric Worst-Case Execution Time Analysis. In *Proceedings of the 3rd International Workshop on Worst-Case Execution Time Analysis, WCET 2003 - a Satellite Event to ECRTS 2003, Polytechnic Institute of Porto, Portugal, July 1, 2003*, Jan Gustafsson (Ed.), Vol. MDH-MRTC-116/2003-1-SE. Department of Computer Science and Engineering, Mälardalen University, Box 883, 721 23 Västerås, Sweden, 99–102.

Francesco Logozzo and Manuel Fähndrich. 2008. On the Relative Completeness of Bytecode Analysis Versus Source Code Analysis. In *Compiler Construction, 17th International Conference, CC 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4959)*, Laurie J. Hendren (Ed.). Springer, 197–212.

Rüdiger Loos and Volker Weispfenning. 1993. Applying Linear Quantifier Elimination. *Comput. J.* 36, 5 (1993), 450–462.

Kuldeep S. Meel and S. Akshay. 2020. Sparse Hashing for Scalable Approximate Model Counting: Theory and Practice. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science* (Saarbrücken, Germany) *(LICS '20)*. Association for Computing Machinery, New York, NY, USA, 728–741.

Antoine Miné. 2001. A New Numerical Abstract Domain Based on Difference-Bound Matrices. In *Proceedings of the Second Symposium on Programs As Data Objects (PADO '01)*. Springer-Verlag, London, UK, UK, 155–172.

Antoine Miné. 2006. The octagon abstract domain. *High. Order Symb. Comput.* 19, 1 (2006), 31–100.

David Monniaux. 2009. Automatic Modular Abstractions for Linear Constraints. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. ACM, New York, NY, USA, 140–151.

David Monniaux. 2010. Quantifier Elimination by Lazy Model Enumeration. In *Proceedings of the 22nd International Conference on Computer Aided Verification* (Edinburgh, UK) *(CAV'10)*. Springer-Verlag, Berlin, Heidelberg, 585–599.

Alexander Nadel and Vadim Ryvchin. 2016. Bit-vector optimization. In *Proceedings of the 22Nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9636 (TACAS'16)*. Springer-Verlag New York, Inc., New York, NY, USA, 851–867.

Nina Narodytska and Fahiem Bacchus. 2014. Maximum satisfiability using core-guided MaxSAT resolution. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence* (Québec City, Québec, Canada) *(AAAI'14)*. AAAI Press, 2717–2723.

Reuven Naveh and Amit Metodi. [n. d.]. Beyond Feasibility: CP Usage in Constrained-Random Functional Hardware Verification. In *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8124)*, Christian Schulte (Ed.).

Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 89–100.

Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark W. Barrett, and Cesare Tinelli. 2018. Solving Quantified Bit-Vectors Using Invertibility Conditions. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10982)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer, 236–255.

Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark W. Barrett, and Cesare Tinelli. 2021. Syntax-Guided Quantifier Instantiation. In *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12652)*, Jan Friso Groote and Kim Guldstrand Larsen (Eds.). Springer, 145–163.

Robert Nieuwenhuis and Albert Oliveras. 2006. On SAT modulo theories and optimization problems. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing* (Seattle, WA) *(SAT'06)*. Springer-Verlag, Berlin, Heidelberg, 156–169.

Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 181–198.

Mathias Preiner, Aina Niemetz, and Armin Biere. 2017. Counterexample-Guided Model Synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10205)*, Axel Legay and Tiziana Margaria (Eds.). 264–280.

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.

Thomas Reps and Gogul Balakrishnan. 2008. Improved Memory-access Analysis for x86 Executables. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction* (Budapest, Hungary) *(CC'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 16–35.

Thomas W. Reps, Shmuel Sagiv, and Greta Yorsh. 2004. Symbolic Implementation of the Best Transformer. In *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, Italy, January 11-13, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 2937)*, Bernhard Steffen and Giorgio Levi (Eds.). Springer, 252–266.

Thomas W. Reps and Aditya V. Thakur. 2016. Automating Abstract Interpretation. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings (Lecture Notes in Computer Science, Vol. 9583)*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer, 3–40.

Fabian Ritter. 2015. *Compiler Optimizations using Symbolic Abstraction.* Ph. D. Dissertation. Saarland University.

Sriram Sankaranarayanan, Franjo Ivančić, Ilya Shlyakhter, and Aarti Gupta. 2006. Static Analysis in Disjunctive Numerical Domains. In *Proceedings of the 13th International Conference on Static Analysis* (Seoul, Korea) *(SAS'06)*. Springer-Verlag, Berlin, Heidelberg, 3–17.

Roberto Sebastiani and Silvia Tomasi. 2015a. Optimization Modulo Theories with Linear Rational Costs. *ACM Trans. Comput. Logic* 16, 2, Article 12 (Feb. 2015), 43 pages.

Roberto Sebastiani and Silvia Tomasi. 2015b. Optimization modulo theories with linear rational costs. *ACM Trans. Comput. Logic* 16, 2, Article 12 (Feb. 2015), 43 pages.

Roberto Sebastiani and Patrick Trentin. 2015a. OptiMathSAT: A Tool for Optimization Modulo Theories. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9206)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 447–454.

Roberto Sebastiani and Patrick Trentin. 2015b. Pushing the envelope of optimization modulo theories with linear-arithmetic cost functions. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9035*. Springer-Verlag, Berlin, Heidelberg, 335–349.

Tushar Sharma, Thomas Reps, and Aditya Thakur. 2013. *An Abstract Domain for Bit-Vector Inequalities.* Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

Tushar Sharma and Thomas W. Reps. 2017. Sound Bit-Precise Numerical Domains. In *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10145)*, Ahmed Bouajjani and David Monniaux (Eds.). Springer, 500–520.

Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society, 138–157.

Axel Simon and Andy King. 2007. Taming the Wrapping of Integer Arithmetic. In *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4634)*, Hanne Riis Nielson and Gilberto Filé (Eds.). Springer, 121–136.

Gagandeep Singh, Markus Püschel, and Martin Vechev. 2017a. Fast Polyhedra Abstract Domain. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. ACM, New York, NY, USA, 46–59.

Gagandeep Singh, Markus Püschel, and Martin Vechev. 2017b. A Practical Construction for Decomposing Numerical Abstract Domains. *Proc. ACM Program. Lang.* 2, POPL, Article 55 (Dec. 2017), 28 pages.

Gagandeep Singh, Markus Püschel, and Martin Vechev. 2017c. A Practical Construction for Decomposing Numerical Abstract Domains. *Proc. ACM Program. Lang.* 2, POPL, Article 55 (Dec. 2017), 28 pages.

Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2015. Making numerical program analysis fast. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Steve Blackburn (Eds.). ACM, 303–313.

Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2018. Fast Numerical Program Analysis with Reinforcement Learning. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10981)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer, 211–229.

Jia Song and Jim Alves-Foss. 2016. The DARPA Cyber Grand Challenge: A Competitor's Perspective, Part 2. *IEEE Secur. Priv.* 14, 1 (2016), 76–81.

Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society.

Allison Sullivan, Darko Marinov, and Sarfraz Khurshid. 2019. Solution Enumeration Abstraction: A Modeling Idiom to Enhance a Lightweight Formal Method. In *Formal Methods and Software Engineering - 21st International Conference on Formal Engineering Methods, ICFEM 2019, Shenzhen, China, November 5-9, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11852)*, Yamine Aït Ameur and Shengchao Qin (Eds.). Springer, 336–352.

Aditya V Thakur, Matt Elder, and Thomas W Reps. 2012. Bilateral Algorithms for Symbolic Abstraction.. In *Proceedings of the 19th International Conference on Static Analysis* (Deauville, France) *(SAS'12)*. Springer-Verlag, Berlin, Heidelberg, 111–128.

Aditya V. Thakur, Akash Lal, Junghee Lim, and Thomas W. Reps. 2015. PostHat and All That: Automating Abstract Interpretation. *Electron. Notes Theor. Comput. Sci.* 311 (2015), 15–32.

Aditya V. Thakur and Thomas W. Reps. 2012. A Method for Symbolic Computation of Abstract Operations. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings (Lecture Notes in Computer Science, Vol. 7358)*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer, 174–192.

Shiyi Wei, Piotr Mardziel, Andrew Ruef, Jeffrey S. Foster, and Michael Hicks. 2018. Evaluating Design Tradeoffs in Numeric Static Analysis for Java. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer, 653–682.

Christoph M Wintersteiger, Youssef Hamadi, and Leonardo De Moura. 2013. Efficiently solving quantified bit-vector formulas. *Form. Methods Syst. Des.* 42, 1 (Feb. 2013), 3–23.

Bo-Han Wu and Chung-Yang (Ric) Huang. 2013. A robust constraint solving framework for multiple constraint sets in constrained random verification. In *The 50th Annual Design Automation Conference 2013, DAC '13, Austin, TX, USA, May 29 - June 07, 2013*. ACM, 119:1–119:7.

Greta Yorsh, Thomas W. Reps, and Shmuel Sagiv. 2004. Symbolically Computing Most-Precise Abstract Operations for Shape Analysis. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 2988)*, Kurt Jensen and Andreas Podelski (Eds.). Springer, 530–545.

Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) *(SEC'18)*. USENIX Association, Berkeley, CA, USA, 745–761.