ASE: A Value Set Decision Procedure for Symbolic Execution

Alireza S. Abyaneh
Department of Computer Sciences
University of Salzburg
Salzburg, Austria
alireza.abyaneh@cs.uni-salzburg.at

Christoph M. Kirsch
Department of Computer Sciences
University of Salzburg, Salzburg, Austria
and *Czech Technical University*, Prague, Czech Republic
christoph.kirsch@cs.uni-salzburg.at

***Abstract*—A symbolic execution engine regularly queries a Satisfiability Modulo Theory (SMT) solver to determine reachability of code during execution. Unfortunately, the SMT solver is often the bottleneck of symbolic execution. Inspired by abstract interpretation, we propose an abstract symbolic execution (ASE) engine which aims at querying the SMT solver less often by trying to compute reachability faster through an increasingly weaker abstraction. For this purpose, we have designed and implemented a value set decision procedure based on strided value interval (SVI) sets for efficiently determining precise, or under-approximating value sets for variables. Our ASE engine begins reasoning with respect to the SVI abstraction, and then only if needed uses the theory of bit-vectors implemented in SMT solvers. Our ASE engine efficiently detects when the former abstraction becomes incomplete to move on and try the next abstraction.**

We have designed and implemented a prototype of our engine for a subset of 64-bit RISC-V. Our experimental evaluation shows that our prototype often improves symbolic execution time by significantly reducing the number of SMT queries while, whenever the abstraction does not work, the overhead for trying still remains low.

***Index Terms*—symbolic execution, value set decision procedure, strided value interval set abstraction**

I. Introduction

Symbolic execution [1]–[4] computes inputs to a given program that make the program run into an error state such as division by zero within a given number of steps during concrete execution. For this purpose, a symbolic execution engine constructs, during program execution, a Satisfiability Modulo Theory (SMT) formula for a given program path to a control-flow location, called path condition, that is satisfiable in the theory of program expressions if and only if the location is reachable on that path. A variable assignment that satisfies the SMT formula corresponds to an input that makes the program run into the location on that path. Conversely, an unsatisfiable SMT formula indicates that the location is unreachable for all inputs that take the program onto that path. The engine regularly queries an SMT solver to determine reachability of the execution branches appearing in the program. Despite recent advances, constraint solving is a scalability bottleneck in symbolic execution of code [1]. Employing a lightweight reasoning procedure to make decisions about the reachability of each of the execution branches enables the symbolic execution engine to scale and to penetrate deeper into the code.

While symbolically executing code, the generated constraints are typically modeled in the theory of bit-vectors [5] to be passed to an SMT solver. A constraint solver for the theory of bit-vectors with a solving algorithm which is designed to operate on an arbitrarily general set of constraints including a wide range and combination of operations may be inefficient for a set of constraints which use a small number of operations and have specific features [6], [7]. In this paper we propose an abstract symbolic execution (ASE) engine exploiting an integer decision procedure for constraint solving which performs a theory integration in a layered manner by considering two abstractions: strided value interval (SVI) set abstraction [8], [9] and bit-vectors [5]. The respective decision procedures for each abstraction in this design strategy are organized in a layered order of increasing capability and complexity. All the decisions are made without querying the SMT solver for the set of constraints which can be solved using the SVI decision procedure in the first layer. Otherwise, the theory of bit-vectors implemented in SMT solvers is responsible for answering reachability queries.

The decision procedure in the first layer employs a lightweight value set analysis technique designed for symbolic execution which uses the SVI abstraction to propagate values and speedup the process of reachability decision making. The SVI abstraction uses a set of strided value intervals to specify values of program variables precisely. An SVI in this set is represented by a wrapped strided interval [8]–[12] which maintains an incrementing step of possible values in addition to the value bounds and allows wrapping in case of overflow. Keeping the incrementing step enables us to reason about multiplication which introduces steps in possible values. By employing the SVI abstraction the decision procedure analyzes the constraints at a higher level than bits (e.g., words and double words) in contrast to bit-precise reasoning used in typical SMT solvers which may be inefficient because of costly *bit-blasting* [5] in their backend [6], [7].

The strided intervals and value set analysis technique are typically used in the context of static analysis to over-approximate the values of variables in the program [8]–[12]. However, in the context of symbolic execution the precision of the decisions over reachability matters. Therefore, the SVI abstraction in ASE is used to represent values that variables

can certainly take (no false positive). Such a value set enables us to make reachability or unreachability decisions precisely. The ASE engine is aware of when the SVI decision procedure is able to provide precise decisions over the set of underlying constraints, and when this cannot be achieved because of a high imposing complexity or incapability, the baseline abstraction is upgraded to bit-vectors.

When the representation of precise values for variables is not possible using the SVI abstraction, we propose a technique in the SVI abstraction layer that under-approximates the set of possible values for involving variables in an operation when applicable. The technique employs an incomplete, lightweight extension of the propagation technique used for our precise SVI decision procedure to make satisfiability decisions and reduce the number of queries sent to the SMT solver and speed up the symbolic execution time.

The following contributions are made by this paper: 1) We propose an ASE engine which employs a theory integration in the constraint solving component of the symbolic execution process, and benefits from a novel value set analysis technique to propagate values at a higher level than bits. 2) We propose a value set decision procedure based on the SVI abstraction, define the theory behind it, and provide conditions under which the analysis provides precise values for program variables. As such, the decision procedure detects when such an abstraction leads to exact satisfiability and unsatisfiability decisions for symbolic execution (Section IV). 3) In order to benefit from the efficiency of our precise SVI decision procedure we introduce an extension technique which enables the ASE engine to make satisfiability decisions using an under-approximation of the set of possible values for variables (Section V).

Our experimental evaluation on a set of benchmarks shows on average 33.62% and 59.84% reduction in symbolic execution time compared to the state-of-the-art approach [13] and the approach which always queries an SMT solver, respectively.

II. Overview

The principled idea of abstract symbolic execution is to integrate symbolic execution with decision procedures that leverage domain specific reasoning to speedup constraint solving by employing increasingly weaker abstractions. The decision procedure we propose here is able to reason about the set of constraints generated out of a program with respect to an abstract domain:

Definition 2.1: A constraint satisfaction problem on finite domains (CSP) is defined by a triplet $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ where $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a set of variables, $\mathcal{D} = \{d_1, \ldots, d_n\}$ is the set of variables' domains ($x_i \in d_i$), and $\mathcal{C} = \{c_1, \ldots, c_m\}$ is a set of constraints over variables. A valuation is defined as $v = (v_1, \ldots, v_n)$ where $v_i \in d_i$. A solution to the CSP is defined as a valuation s which satisfies the set of constraints $\mathcal{C}$. The set of all solutions is denoted as $\mathcal{S}$.

Given a specific abstract domain to represent the possible values of variables, the set of all solutions $\mathcal{S}$ for a set of constraints $\mathcal{C}$ can be represented precisely, or approximated.

Definition 2.2: Given an abstract domain $\mathcal{A}$ and a corresponding set of solutions $\mathcal{S}^{\mathcal{A}}$, this set is precise if $\mathcal{S}^{\mathcal{A}} = \mathcal{S}$ and is an under-approximation if $\mathcal{S}^{\mathcal{A}} \subseteq \mathcal{S}$. In case of an under-approximation we write $\underline{\mathcal{S}}^{\mathcal{A}}$ for $\mathcal{S}$.

In this paper we present a decision procedure employing a propagation technique which considers an SVI set as its underlying abstract domain to find the precise, or the under-approximating set of all solutions to a CSP. To do so, we employ a refinement model [14] wherein initially variables can take any value in their domain. While symbolically executing code, constraints are generated and gradually variables' values are refined so that they satisfy the currently seen set of constraints. When the SVI decision procedure cannot provide a decision, the abstraction is upgraded to bit-vectors and the decision procedure implemented in an SMT solver is used to make a decision.

Section IV presents the precise SVI decision procedure and discusses the conditions under which the analysis provides the exact set of all solutions. In Section V we propose a technique which continues propagating an under-approximation of the set of solutions when representation of the precise solutions is not possible. Keeping the under-approximating solutions $\underline{\mathcal{S}}^{\mathcal{A}}$ for the currently seen constraints enables us to check the satisfiability (and not unsatisfiability) of subsequent constraints appearing in the program while still using the efficient propagation technique of the decision procedure of Section IV. The proposed symbolic execution algorithm and its implementation details are explained in Section VI.

III. Related Work

In this paper we employ an abstract domain in the process of symbolic reasoning about the set of generated constraints out of a program. Abstracting values using interval domain is a well-studied and efficient technique in the context of abstract interpretation and program analysis. The concept of strided intervals was introduced by Reps et al. [10] representing fixed-precision value intervals with incrementing steps. However, a strided interval in their definition does not permit wrapping [15]. The application of the interval abstract domain becomes harder when it comes to the analysis of machine arithmetic with wrap-around semantics in case of overflow. Integer wrapped intervals [15], [16] extends the classical intervals by considering wrap-around semantics. Gange et al. [15] provide a sound analysis for wrapped interval abstract domain. They use intervals to over-approximate the set of all possible values which one variable can take. In our SVI decision procedure we use their soundness conditions, however we extend the abstraction to wrapped strided intervals, propose completeness conditions for operations, and define the detailed semantics of reachability decision making when the comparison operation is involved. In particular we define the inverse semantics of operations while propagation intervals backward (Section IV-B).

Sen et al. [8] employ the wrapped interval abstraction with steps where they propose a new abstract domain called Circular Linear Progressions (CLPs) to combine the efficiency of interval abstract domain with the discreteness of linear

congruence domain. In our SVI decision procedure we use their defined semantics, however we extend those by defining soundness and completeness conditions for each operation since in the context of symbolic execution the precision of decisions matters. Moreover, we discuss the inverse semantics of operations while propagating constraints backward (Section IV-B), and investigate the semantics of the remainder operation during the forward propagation (Section IV-A).

The strided interval abstraction [10] has been used in several research works [9], [11]. Balakrishnan et al. [11] propose value set analysis (VSA) which uses a combined numeric and pointer analysis algorithm to over-approximate the set of values. Later, to increase the precision, Shoshitaishvili et al. [9] extends the VSA of Balakrishnan et al. [11] to detect memory corruptions by developing a new abstraction called *strided interval set*, implementing a lightweight algebraic solver, and adopting a signedness-agnostic wrapped interval domain. However, they omit discussing the detailed semantics of their algebraic solver. Both papers use VSA as a static analysis technique to over-approximate possible values that registers or abstract locations in a program can take. We also use strided interval sets to analyze possible values, however since we employ our SVI decision procedure in symbolic execution, it is necessary to be able to make exact decisions. Therefore, additionally we derive completeness conditions under which our analysis provides exact values for variables and leads to exact reachability decisions. Moreover, the provided under-approximating values by ASE can also lead to exact reachability decisions.

The layering design strategy which refers to employing a set of theory solvers in an increasing order of capability and complexity is another related research topic [17]–[19]. Bruttomesso et al. [17] proposed a three-layered theory solver including a solver for the logic of equality of uninterpreted functions as first layer, an incomplete solver which is based on term rewriting technique and inconsistency detection for bit-vector constraints as second layer, and a complete solver for bit-vectors as third layer. Hadarean et al. [18] proposed a lazy, layered approach for the theory of bit-vectors targeting equality and inequality constraints. Their proposed inequality solver applies a polynomial-time algorithm to check satisfiability given a set of constraints. In this paper we also use a layering design strategy in our decision procedure. However, we propose an SVI decision procedure as the first layer.

In the context of symbolic execution there are many tools which implement the symbolic execution algorithm targeting different languages and by applying different techniques [1]–[4], [20]–[24]. In this paper we extend the idea of PARTI [13] which proposed an optimization technique to reduce the number of accesses to the constraint solver by implementing an incomplete solver for interval theory. Variable's values in such a solver are represented by a data-structure called *multi-interval*. The expressions and operations which can be handled by this incomplete solver are solved efficiently and the rest of the queries are passed to a complete SMT solver. The contributions of this paper over PARTI [13] are as follows. 1) we extend the employed abstraction in PARTI, which is a set of non-wrapped value intervals, to a set of wrapped strided value intervals. The strided value intervals introduce an incrementing step for possible values needed to reason about linear multiplication. 2) Given the SVI abstraction, we propose the corresponding propagation algorithm by defining the precise, formal semantics of each operation together with soundness and completeness conditions (Section IV). The extraction of such conditions is essential to make exact reachability decisions. PARTI omits to define the theory behind its engine formally including the detailed semantics of the operations. 3) When a complete analysis is possible, ASE extents the theory by supporting several more operations including linear multiplication, division, and remainder. 4) We propose an optimization technique based on the SVI abstraction which aims at solving satisfiability by keeping an under-approximating set of values for variables (Section V). On the other hand, PARTI supports features that ASE does not such as signed integer arithmetic and type casting.

IV. Precise SVI Decision Procedure

In this section we present the value set decision procedure which is used in the ASE engine with respect to the SVI abstraction. Listing 1 shows a code example where a symbolic input is provided to the program. The execution of the program at line 2 assigns a symbolic value which varies in a range from 10 to 20 into variable x. Then, at line 3 variable y is assigned through a symbolic expression which therefore assigns values $\{20, 22, \ldots, 40\}$ to y. These values cannot be represented by only bounds because some values in between are missing. In order to handle such a case which is the result of a multiplication we also need to keep the incrementing step of possible values in addition to the bounds, denoted as a tuple $\langle lo, up, step \rangle$. Thus, the possible values for y are represented as $\langle 20, 40, 2 \rangle$. The assignment of an arithmetic expression to a variable is evaluated by the substitution of variables appearing on the right hand side. We call this process *forward propagation of constraints*. Afterwards, the execution of the conditional expression at line 4 creates two execution branches. The evaluation of the branch to true and false, tightens the possible values of y to $\langle 20, 28, 2 \rangle$ and $\langle 30, 40, 2 \rangle$, respectively. Since the assignment at line 3 created an alias relation between variables y and x, the values for x have to be updated in consequence of the tightened values for y. Therefore, the new values for x on the true and false execution paths are $\langle 10, 14, 1 \rangle$ and $\langle 15, 20, 1 \rangle$, respectively. As it is manifested in this example, the evaluation of a conditional expression may need to update previously stored value intervals for involving variables and those which were in relation with them through an assignment (which creates memory aliasing). We call this process *backward propagation of constraints*. The value intervals of variable x at each endpoint is used to generate test inputs which trigger each of two execution paths in Listing 1.

```
int x, y;                        1
interval(&x, [10, 20]);          2
y = 2 * x;                       3
if (y < 30) {...;}               4
else {...;}                      5
```

Listing 1: Code example.

In the context of fixed-width machine arithmetic with overflow semantics, integer variables can only take a finite range of values representable by w bits. The performed operations on these integers are denoted by a subscript w (e.g., $+_w$). In this paper we assume all arithmetic operations are unsigned and for integers. In fact integer values are represented by a sequence of bits of fixed length and signedness is an interpretation of those bits by using a method like two's complement to represent signed numbers.

Definition 4.1: (Strided Value Interval). A strided value interval (SVI) with wrap-around semantics is a tuple $\langle a, b, s\rangle_w$ where $s > 0$, each of a, b, and s are represented by w-bit integers, and $s \mid (b -_w a)$. The tuple indicates the set of integer values $\{a +_w i * s \mid 0 \le i \le i_{max}\}$ where i_{max} is the smallest integer value which satisfies $a +_w i_{max} * s = b$. The elements a, b, and s specify the lower bound, upper bound, and stride (or step) of the values.

Definition 4.2: (Cardinality). The cardinality of a SVI denoted as $card(\langle a, b, s\rangle_w)$ is the number of integer values in that interval and it is computed as $(b -_w a)/s +_w 1$ where $/$ is an integer division.

Definition 4.3: (Strided Value Interval Set). A strided value interval set represents a set of SVIs with an identical stride indicating the values that a variable in the program can take.

Following the wrap-around semantics the bounds for a range of values are represented on a number circle (as opposed to number line) [8], [15]. The values for an unsigned integer start with 0 and proceed clockwise to the maximum value which is $2^w - 1$. The representation of numbers in a circular manner enables us to depict the overflow concept. For example, the addition of maximum integer value, i.e. $2^w - 1$, and 1 results in 0 when a variable is represented by w bits. Given a value interval $\langle a, b, s\rangle_w$, in the circular representation the starting point of the interval can take place in a position where either $a \le b$ or $a > b$. The latter case specifies the occurrence of an overflow which indicates a wrapping. Interval $\mathcal{I} = \langle a, b, s\rangle_w$ represented on a number circle can be decomposed into at most two sub-intervals which do not wrap as follows:

$$\begin{cases} \langle a, b, s\rangle_w \hookrightarrow \langle a, b, s\rangle_w & a \le b \\ \langle a, b, s\rangle_w \hookrightarrow \langle a, max, s\rangle_w \cup \langle min, b, s\rangle_w & otherwise \end{cases} \tag{1}$$

where $min = glb_{\mathcal{I}}(0)$ and $max = lub_{\mathcal{I}}(2^w - 1)$. The $lub_{\mathcal{I}}(i)$ function maps its argument i to the largest value that belongs to the interval $\mathcal{I} = \langle a, b, s\rangle_w$ and is less-than or equal to i. The $glb_{\mathcal{I}}(i)$ function maps i to the smallest value that belongs to the interval $\mathcal{I} = \langle a, b, s\rangle_w$ and is greater-than or equal to i.

The multi-interval data structure used in PARTI [13] considers a set of non-wrapped intervals which only keeps value bounds. It always splits the values when an overflow occurs. Moreover, representation of values with an incrementing step, may not be efficient using the multi-interval data structure. For example, considering unsigned 64-bit integers, the set $\{2^{64} - 1, 2, 5\}$ contains 3 values where $2^{64} - 1$ is the maximum representable value for an unsigned 64-bit integer. Given the multi-interval abstraction this set is represented as $\{[2^{64} - 1, 2^{64} - 1], [2, 2], [5, 5]\}$, whereas using the SVI abstraction this can be represented by $\langle 2^{64} - 1, 5, 3\rangle_{64}$.

In the rest of this section we define the semantics for forward and backward propagation of an SVI. Since we design a decision procedure for symbolic execution, it is critical to extract conditions under which the analysis provides correct and exact decisions based on the employed SVI abstraction.

A. Forward Propagation of Constraints

This section presents the forward semantics of arithmetic operations and specifies the conditions under which the resulting value interval(s) represent the exact possible values.

1) Addition/Subtraction: The addition and subtraction of a non-empty interval $\langle a, b, s\rangle_w$ and a concrete integer value k are computed as follows:

$$\begin{aligned} \langle a, b, s\rangle_w + k &= \langle a +_w k, b +_w k, s\rangle_w \\ \langle a, b, s\rangle_w - k &= \langle a -_w k, b -_w k, s\rangle_w \\ k + \langle a, b, s\rangle_w &= \langle k +_w a, k +_w b, s\rangle_w \\ k - \langle a, b, s\rangle_w &= \langle k -_w b, k -_w a, s\rangle_w \end{aligned} \tag{2}$$

Given two value intervals $\langle a, b, s\rangle_w$ and $\langle c, d, s'\rangle_w$ where cardinality of each is greater than 1, the addition and subtraction of those are computed as follows [8]:

$$\begin{aligned} \langle a, b, s\rangle_w + \langle c, d, s'\rangle_w &= \langle a +_w c, b +_w d, gcd(s, s')\rangle_w \\ \langle a, b, s\rangle_w - \langle c, d, s'\rangle_w &= \langle a -_w d, b -_w c, gcd(s, s')\rangle_w \end{aligned} \tag{3}$$

where gcd computes the greatest common divisor of two integers. The resulting interval computed in Formula 3 is a sound over-approximation of all possible resulting values when the following condition is satisfied [15]:

$$card(\langle a, b, 1\rangle_w) + card(\langle c, d, 1\rangle_w) \le 2^w + 1 \tag{4}$$

which means that the length of the resulting interval, i.e. $(b - a) + (d - c)$, should not be overflowed. For the case wherein both operands are symbolic and the condition in Formula 4 is satisfied, Formula 3 may provide an interval which is a superset of the exact possible values in two following cases. First case may happen when the steps of two intervals corresponding to the operands are not identical. For example, the addition of $l = \langle 0, 90, 10\rangle_{64}$ and $r = \langle 0, 7, 1\rangle_{64}$ value intervals results a proper subset of $\langle 0, 97, 1\rangle_{64}$ containing values such as 88 which cannot be inferred from $l + r$. We derive a condition under which the resulting interval in Formula 3 can provide the exact resulting values:

$$\begin{aligned} &s' \,\%\, s = 0 \land card(\langle a, b, s\rangle_w) \ge s'/s \quad \text{or} \\ &s \,\%\, s' = 0 \land card(\langle c, d, s'\rangle_w) \ge s/s' \end{aligned} \tag{5}$$

The second case is when the operands are related to each other, for example $x+x$ or $x-x$. In such a case the resulting interval will be an over-approximation of possible values.

2) Multiplication: The multiplication of a non-empty interval $\langle a, b, s\rangle_w$ and a concrete integer value $k > 0$, is computed as:

$$\langle a, b, s\rangle_w * k = \langle a *_w k, b *_w k, s * k\rangle \tag{6}$$

The conditions under which the above formula provides an exact resulting values are defined as follows: $(b - a) * k < 2^w$ and $s * k < 2^w$, which implies that the length of the resulting interval after multiplication [15] and the resulting

step must not be overflowed. PARTI [13] supports left shifts by using a so-called left shift attribute in its decision stage. Otherwise, non-constant multiplications are not supported by PARTI since it introduces value steps.

3) Division: Given a non-empty interval $\langle a, b, s\rangle_w$ and a concrete integer value $k > 0$, the division $\langle a, b, s\rangle_w \,/\, k$ is computed as follows:

$$\begin{cases} \langle a\,/\,k, b\,/\,k, s_r\rangle_w & a \leq b \\ \langle min\,/\,k, max\,/\,k, s_r\rangle_w & a > b \end{cases} \tag{7}$$

where min and max have the same definition as in Formula 1. s_r is computed as $s\,/\,k$ when $s \geq k$ and otherwise is equal to 1. Formula 7 provides a sound over-approximation of possible values as result of division. In order to derive the resulting interval which represents the exact possible values, we need to specify two conditions. First when $a \neq b$, by checking whether the step s and the divisor k are divisible or not which means: $s \,\%\, k = 0$ if $s \geq k$. The second condition applies on when $a > b$. In this case the interval $\langle a, b, s\rangle_w$ should be split into two sub-intervals $\langle a, max, s\rangle_w$ and $\langle min, b, s\rangle_w$ according to Formula 1. The resulting values can be represented as the two following intervals: $\langle a\,/\,k, max\,/\,k, s_r\rangle$ and $\langle min\,/\,k, b\,/\,k, s_r\rangle$. PARTI [13] only supports right shifts by constant.

4) Remainder: Given a non-empty interval $\langle a, b, s\rangle_w$ where $a \leq b$ and a concrete integer value $k > 0$, the remainder operation $\langle a, b, s\rangle_w \,\%\, k$ can be computed as:

$$\begin{cases} \langle a\%k, b\%k, s\rangle_w & b/k = a/k \\ \langle a\%k - i_{start} * gcd(s,k), \\ \quad a\%k + i_{end} * gcd(s,k), & b - a \geq lcm(s,k) - s \,\wedge \\ \quad gcd(s,k)\rangle_w & b/k \neq a/k \\ \langle 0, k-1, 1\rangle_w & otherwise \end{cases} \tag{8}$$

wherein lcm is the least common multiple function, $i_{start} = (a\%k)/\,gcd(s,k)$, and $i_{end} = (k - 1 - a\%k)/gcd(s,k)$. All the arithmetic operations in the above formula are modulo 2^w. When one of the two first conditions in Formula 8 are satisfied, the resulting interval represents the exact values. However in case of violation, the *otherwise* case represents an over-approximation of the possible values.

In case of $a > b$, the resulting values should be the remainder of both k and 2^w. When $b - a \geq lcm(s,k) - s$ and $gcd(k, 2^w) = k$ then the resulting values can be represented by the second rule in Formula 8. Otherwise if the conditions are not satisfied, interval $\langle a, b, s\rangle_w$ should be split into two sub-intervals according to Formula 1 and each is analyzed separately. The remainder operator is not supported by PARTI [13].

5) Comparison: The unsigned comparison expressions, $x \,\texttt{cmp}\, y$, where $\texttt{cmp} \in \{<, \leq, >, \geq, =, \neq\}$ can all be converted into an equivalent expression using only $<$ operation. For operations $\texttt{cmp} \in \{\leq, >, \geq, =, \neq\}$ the equivalent versions are $1 - (y < x)$, $y < x$, $1 - (x < y)$, $y - x < 1$, and $0 < y - x$, respectively. Therefore, we only need to

$x \overset{?}{<} y$	Evaluation	Resulting Interval	
		x	y
if $b < c$	$false$	$\bot$	$\bot$
	$true$	$\langle a, b, s_x\rangle$	$\langle c, d, s_y\rangle$
if $d \leq a$	$false$	$\langle a, b, s_x\rangle$	$\langle c, d, s_y\rangle$
	$true$	$\bot$	$\bot$
if $c = d \,\wedge$ $a < c \leq b$	$false$	$\langle c, b, s_x\rangle$	$\langle c, d, s_y\rangle$
	$true$	$\langle a, lub_x(c{-}1), s_x\rangle$	$\langle c, d, s_y\rangle$
if $a = b \,\wedge$ $c \leq a < d$	$false$	$\langle a, b, s_x\rangle$	$\langle c, a, s_y\rangle$
	$true$	$\langle a, b, s_x\rangle$	$\langle glb_y(a{+}1), d, s_y\rangle$

TABLE I: The exact evaluation of less-than operation.

reason about $<$ operation. Given two intervals $x = \langle a, b, s_x\rangle$ and $y = \langle c, d, s_y\rangle$ where $a \leq b$ and $c \leq d$, Table I shows the conditions under which the evaluation of $x < y$ can be decided by specifying the exact values that each operand can take.

If the intervals for both operands are wrapped then the precise operands' values cannot be efficiently computed. However, if only one of them is wrapped and the other operand is a concrete value then a precise decision can be made. In this case the evaluation is done by splitting the wrapped interval into two sub-intervals based on Formula 1 and then comparing the operands accordingly.

Moreover, if either of the intervals for operands are represented by more than one SVI, the other operand has to be a concrete value so that exact decisions can be made.

B. Backward Propagation of Constraints

Taking each branch of a conditional expression might tighten the bounds of possible values for each operand. In consequence, all the other variables in the program which were in relation with those operands through an assignment expression and up to the current program location have to be updated. For example, if we consider the interval $\langle 10, 20, 1\rangle$ for variable x and execute $y = 2 * x$ the resulting interval for y is $\langle 20, 40, 2\rangle$. Now the program may contain either $if(y < 30)$ or $if(x < 18)$ statements which forms two different types of bound propagation. The first one should revise the value intervals back to the variables from which y was computed (i.e., x through the expression $y = 2 * x$), and the second one should update resulting variables of the assignment expressions in which x were involved (i.e., y in the expression $y = 2 * x$). We call these two types of backward propagation as *left-to-right*, and *right-to-left* propagations, respectively. In our example the condition $if(y < 30)$ tightens the constraint on y to $\langle 20, 28, 2\rangle$ and then this new constraint should be propagated to update the value interval of x to $\langle 10, 14, 1\rangle$. Furthermore, for the right-to-left propagation type, the condition $if(x < 18)$ tightens the constraint on x to $\langle 10, 17, 1\rangle$ and then this new interval is propagated through the expression $y = 2 * x$ which updates the value interval of y to $\langle 20, 34, 2\rangle$.

The right-to-left propagation is the same as what is done in forward propagation of constraints as described in Section IV-A. In order to deal with the left-to-right propagation we need to define the inverse semantics of each operation.

Let us assume that the values for variable x are represented as $\langle a, b, s_x\rangle$. If we execute the expression $y = x \odot k$ where

$\odot \in \{+,-,*,/,\%\}$ and $k > 0$ is a concrete integer value, the resulting values for y will be represented by $\langle c, d, s_y\rangle$. Now the execution of the conditional expression $if(y<t)$ may tighten the possible values of y to $\langle c', d', s_y\rangle$ where $c', d' \in \langle c, d, s_y\rangle$. In order to update the values of x based on the new values of y, we need to inverse the applied operation as follows.

1) Addition/Subtraction: The addition can be reversed by subtraction and the subtraction is reversed using either addition or subtraction. Table II shows how to reverse an addition or a subtraction by indicating expressions to compute bounds of variable x. The semantics of addition and subtraction to do reversion is the same as what is explained in Section IV-A.

Operation	New Bounds
$x+k$ or $k+x$	$a' = c' - k$ $b' = d' - k$
$x-k$	$a' = c' + k$ $b' = d' + k$
$k-x$	$a' = k - d'$ $b' = k - c'$

TABLE II: Inverse of addition and subtraction operations.

2) Multiplication: The multiplication can be reversed using division as follows:

$$a' = a + (c' - c)/k \quad \text{and} \quad b' = a + (d' - c)/k \tag{9}$$

for example, considering $\langle 2^{63}, 2^{64}-1, 1\rangle_{64}$ as the value interval of x, then multiplication of x by 2 results in $\langle 0, 2^{64}-2, 2\rangle_{64}$ when the arithmetic is modulo 2^{64}. Now we execute $if(y < 2^{32})$ and thus the tightened interval for y is $\langle 0, 2^{32}-2, 2\rangle_{64}$. The original bounds for x can be updated as $2^{63}+(0-0)/2$ and $2^{63}+(2^{32}-2-0)/2$. PARTI [13] omits explaining the details of how multiplication is reversed using a division especially while wrapping occurs.

3) Division: Whenever for the original interval of x condition $a \leq b$ is satisfied, the new bounds for x are as follows:

$$a' = \begin{cases} a & c' * k \leq a \\ glb_x(c' * k) & c' * k > a \end{cases} \\ b' = lub_x(d' * k + \mathit{offset}) \tag{10}$$

Moreover, *offset* is computed as:

$$\begin{cases} b - d' * k & b < d' * k + k - 1 \\ k - 1 & b \geq d' * k + k - 1 \end{cases} \tag{11}$$

In fact, the purpose of *offset* is to reverse the side effect of integer division. In integer division the result loses its decimal part (e.g. $15/2 = 7$) and in order to be able to reverse that we need to compute *offset*. For example, we assume $x = \langle 11, 27, 4\rangle$ and we execute $y = x/2$ then $y = \langle 5, 13, 2\rangle$ and the execution of $if(5 < y)$ results in a new value interval $y = \langle 7, 13, 2\rangle$ and thus $x = \langle 15, 27, 4\rangle$.

In case of $a > b$, inversion should be applied with respect to two non-wrapped sub-intervals $\langle a, max, s_x\rangle$ and $\langle min, b, s_x\rangle$ of x computed according to Formula 1. The backward propagation of division is not supported by PARTI [13].

4) Remainder: To reverse remainder, we need to compute values inside the interval $\langle a, b, s_x\rangle$ for which $x\%k$ results a value in $\langle c', d', s_y\rangle$. The analysis of this operation may lead to a large number of value intervals and thus may be costly using the SVI abstraction. Therefore, we omit to apply the exact backward propagation on remainder operation.

C. Time Complexity

The time complexity of the decision procedure proposed in this section depends on the number of instructions and their cost. Given an execution path in a program, we assume that n indicates the number of instructions on the path, k identifies the maximum number of value intervals returned by applying an operation, and c represents a constant indicating the upper-bound on the cost of applying an operation. While doing forward propagation, the time complexity of each addition or subtraction is $\mathcal{O}(k^2 \cdot c)$. Moreover, multiplication, division, modulo, and comparison operations each can be computed in $\mathcal{O}(k \cdot c)$. Backward propagation of intervals takes $\mathcal{O}(n \cdot k \cdot c)$ since the depth of backward propagation might be proportional to the number of executed instructions on the path. Therefore, the total time complexity of the analysis for an execution path is computed as $\mathcal{O}(n \cdot k^2 \cdot c) + \mathcal{O}(n^2 \cdot k \cdot c)$. Let us assume that a symbolic expression is represented by an abstract syntax tree. Each intermediate node of this tree represents an arithmetic operation and each leaf node represents a concrete or a symbolic input value. When the symbolic expressions representing variables and operands have shallow depth and can be bounded by a constant value d, then this time complexity order tends to $\mathcal{O}(n \cdot k^2 \cdot c) + \mathcal{O}(n \cdot d \cdot k \cdot c)$.

V. Under-Approximating SVI Decision Procedure

In Section IV we explained the conditions under which our SVI abstraction can provide exact decisions over a set of constraints. In order to exploit the efficiency of the decision procedure presented in Section IV, we propose a technique which enables us to still reason about the satisfiability of a set of constraints efficiently for the cases which conditions in Table I are not satisfied anymore. For example, considering the expression $x \leq y$ where $x \in \langle 0, 10, 1\rangle$ and $y \in \langle 2, 10, 1\rangle$, the striped region in Figure 1b illustrates the values for each of x and y which satisfy $x \leq y$. The values of the operands intersect each other on interval $\langle 2, 10, 1\rangle$ and for this intersection area in addition to the value intervals, a less-than-equal relation has to be set between variables. Representing the precise set of values which each of x and y variables can take is inefficient using the SVI abstraction since it may lead to many relative cases and may increase the complexity of the analysis. The interval theory proposed by Dustmann et al. [13] does not support the backward propagation of such constraints to the involving variables. In learning stage of their theory, only binary operations whose one side is a constant is analyzed.

In such cases where the intervals of the operands in a comparison expression overlap each other, the analysis can still be continued by maintaining an under-approximating set of values, $\underline{S}^{\mathcal{A}}$. Keeping a set of values which satisfy the currently seen set of constraints on the path condition, as opposed to only one generated witness by an SMT solver, may help deciding the satisfiability of the constraints which will appear further on the execution path (The unsatisfiability/unreachability decisions cannot be proved using the under-approximation since the set of all values are not available). The satisfiability of a set of constraints which can be determined by

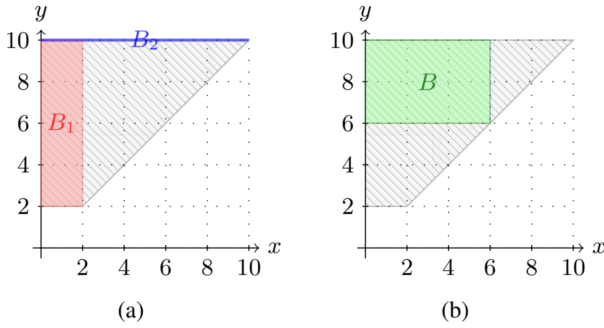

Fig. 1: a) Boxes B_1 and B_2 indicate two candidate under-approximating regions for the evaluation of $x \leq y$ to true. b) Box B indicates an under-approximating region for the evaluation of $x \leq y$ to true.

this set of values does not need to be checked by a complete bit-vector decision procedure (i.e., SMT solver).

Definition 5.1: Given a constraint satisfaction problem $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, a solution box is defined as $\mathcal{B} = (\mathcal{I}_1, \ldots, \mathcal{I}_n)$, where $\mathcal{I}_i \subseteq d_i$ represents a value interval for x_i, satisfying $\mathcal{C}$.

An under-approximating set of possible values can be indicated by a box. Figure 1 shows the region (striped) where $x \leq y$ is evaluated as true where $x \in \langle 0, 10, 1 \rangle$ and $y \in \langle 2, 10, 1 \rangle$. Choosing any box (or set of boxes) in this region determines candidate intervals for each of the operands which under-approximate the possible resulting values. We avoid selecting a set of boxes since it increases the complexity of the propagation algorithm which harms the efficiency. To keep the analysis lightweight, one of the design factors is to trade off accuracy of the resulting candidate box(es) for faster selection of those. We propose and evaluate two methods for the under-approximating box selection based on the following observations. Given the conditional expression $x \leq y$ on an execution path of a program, depending on the further usages of x and y on the path, the selection of different boxes can be useful.

Observation 1. In a program which computes the maximum of elements in an array, the comparison of two array elements x and y always takes the greater variable and continues the execution with that (i.e., y for $x \leq y$, and x for the negation $x > y$). Therefore, the best choice is a box which prioritizes the greater variable. The variable prioritization means that we assign the largest possible value interval satisfying the condition to that variable. The story is reverse while computing the minimum of elements in an array where the comparison of two array elements x and y always takes the smaller variable and continues the execution with that (i.e., y for $x > y$, and x for the negation $x \leq y$). Here the best choice is the box which prioritizes the smaller variable. Based on this observation we propose a heuristic which considers two candidate boxes: the box in which variable y is prioritized over x, and the box in which variable x is prioritized over y. For example, the true evaluation of $x \leq y$, as depicted in Figure 1a, leads to the following choices, respectively: 1) $x = \langle 0, 2, 1 \rangle$ and $y = \langle 2, 10, 1 \rangle$, 2) $x = \langle 0, 10, 1 \rangle$ and $y = \langle 10, 10, 1 \rangle$. We keep the above two choices as candidate boxes for the true evaluation of $x \leq y$ and continue the execution. The first upcoming decision point on the execution path which involves one of the operands will select one of the boxes which can provide a decision and the execution is continued with that choice. For example, if after the true evaluation of $x \leq y$ the execution reaches the condition $y \leq z$ where $z = \langle 0, 5, 1 \rangle$ then for the rest of the execution on this path the box $x = \langle 0, 2, 1 \rangle$ and $y = \langle 2, 10, 1 \rangle$ is chosen for the true evaluation of $x \leq y$ since this choice leads to decisions for both true and false branches of the currently processing condition $y \leq z$.

Observation 2. Given $x \leq y$ on an execution path of a program, the second observation is when we cannot prioritize one variable over the other one because both of them are used further on the execution path. In this case we need to select a range of values for each of the involving variables instead of one concrete value. Therefore, we propose a heuristic which takes the middle point in the intersection of the operands' intervals and derives the respective boxes accordingly. The middle point provides equal chance to each of the operands. Any other choice prioritizes one of the operands over the other one. In Figure 1b the middle point of the intersection of the intervals representing variables x and y is 6, and thus the derived box for the $x \leq y$ will be represented as $x = \langle 0, 6, 1 \rangle$ and $y = \langle 6, 10, 1 \rangle$.

Once the under-approximating box(es) is selected the execution will be continued accordingly. Any satisfiability decision which can be made base on the selected under-approximating value boxes does not need to be checked with the SMT solver.

VI. Implementation Details

This section explains the implementation details of the proposed symbolic execution engine.

Machine State Representation. Symbolic inputs to the program are denoted by set $In = \{\alpha_1, \ldots, \alpha_n\}$ where α_i is represented as a value interval $\mathcal{I}_i$. The symbolic execution engine has to keep track of the machine state transitions as a consequence of executing instructions using symbolic values. The machine state at each step of the execution is represented by the current state of the registers and the memory. The stored values inside the registers and memory addresses may be symbolic or concrete. The stored value is the result of applying binary and unary operations on initial symbolic or concrete input values expressed as a symbolic expression $\mathcal{E}$. The registers and memory states are indicated by a mapping from register numbers and memory addresses to symbolic expressions and values which are guarded by the path condition.

Definition 6.1: A symbolic value is represented as a tuple $\langle \mathcal{E}^{pc}, \mathcal{V}, \mathcal{A} \rangle$ where $\mathcal{E}^{pc}$ specifies the symbolic expression guarded by the path condition, $\mathcal{A}$ indicates the abstraction, and $\mathcal{V}$ denotes the set of values with respect to $\mathcal{A}$.

A symbolic memory address $\mathcal{M}[addr]$, or a symbolic register $\mathcal{R}_i$ is represented by a symbolic value tuple which is associated to the memory address or the register. The element $\mathcal{A}$ of a symbolic value tuple associated to each memory address indicates the abstraction which has been used so far

to reason about the set of constraints involving that memory address. $\mathcal{A}$ mainly takes three values of pvi, $ubox$ and bvt representing precise value interval, under-approximating value box, and bit-vector abstractions. $\mathcal{A} = pvi$ and $\mathcal{A} = ubox$ mean that our SVI decision procedure provides precise, and under-approximating analysis for the involving variable. Otherwise when the theory of bit-vectors is used to reason about the constraints involving that memory address, $\mathcal{A}$ is set to bvt. The value set $\mathcal{V}$ is represented by a set of wrapped strided value intervals which specify the precise or under-approximating values. In the worst-case $\mathcal{V}$ only contains one value which is obtained from the SMT's witness. We use a trace of execution history to keep track of memory state transitions, which is called the *memory trace* data-structure.

Definition 6.2: A trace entry, te, for memory address $addr$ is represented by a tuple $\langle \ell, prev, \mathcal{M} \rangle$ where ℓ is the location of the instruction which caused creating the trace entry, $prev$ points to the previous symbolic value tuple of $addr$, and $\mathcal{M}$ represents the new symbolic value tuple for $addr$.

The operations which update the memory trace are any instructions whose execution affects the symbolic value of a memory address. These operations include storing a symbolic expression into a memory address, evaluating a symbolic conditional expression, and executing a system call such as *read*, or *brk*. At each point of the execution of a program the trace should return the most recently updated symbolic value tuple which is stored into a specific memory address. This is done by keeping track of pointers which assign memory addresses to the corresponding trace entries which contain the most recently updated values. The previously stored values into a memory address by previous instructions and on different pending paths during symbolic execution of code are kept on the trace for further backtracking.

ASE Algorithm. Algorithm 1 illustrates the symbolic execution algorithm. The execution starts at the initial program location ℓ_0. Depending on the instruction which is currently under execution, the engine updates its state. When a memory address (variable) is specified with a symbolic input then a symbolic value tuple representing its value is stored at that memory address (line 5). For example, if we assign a range of values between 10 and 20 to variable a, then this is done by assigning an interval $\langle 10, 20, 1 \rangle$ for our value interval abstraction and a symbolic expression denoting $10 \leq a \leq 20$. A symbolic expression is represented as an abstract syntax tree. A symbolic expression which is evaluated as a concrete value is stored accordingly in the memory.

The content of a memory address $\mathcal{M}[addr]$ is loaded by putting a symbolic value tuple $\langle \mathcal{E}^{pc}, \mathcal{V}, \mathcal{A} \rangle$ stored at that memory address into a destination register (line 7). Each *store*, which may be interpreted as an assignment in high level code, updates the memory with the symbolic value of a register (line 9). Moreover, it keeps track of the memory aliases which is created between variables through this assignment (e.g., the aliasing relation between a and b through $a = 3 * b$).

During the execution whenever a conditional expression $\mathcal{R}_d = \mathcal{R}_i \,\texttt{cmp}\, \mathcal{R}_j$ is reached where $\texttt{cmp} \in \{<, \leq, =, \neq\}$, then

Algorithm 1: Abstract Symbolic Execution Engine

```
   Input  : Program P
 1 ℓ = 0;
 2 while true do
 3   switch op(ℓ) do
 4     case read:
 5       M[addr] = ⟨E_α^pc, V_α, pvi⟩;
 6     case load:
 7       R_i = M[addr];
 8     case store:
 9       M[addr] = R_i;
10     case if(R_d):
         // R_d = R_i cmp R_j,  cmp ∈ {<, <=, =, ≠}
11       false_reachability = evaluate(R_d = false);
12       if false_reachability == true then
13         save_context(R_d = false);
14       end
15       true_reachability = evaluate(R_d = true);
16       if true_reachability == true then
17         pc = pc ∧ (R_d = true);
18       else
19         backtrack();
20       end
21     case end-point:
22       generate_witness();
23       backtrack();
24     default:
25       propagate_forward(op(ℓ), abstraction);
26   end
27   ℓ = next_ℓ;
28   if ℓ = 0 then terminate;
29 end
```

the reachability of the true and false branches is evaluated by calling the `evaluate` function. The reachability of the false branch is tried first and if it is reachable, the symbolic execution context for the false evaluation of the conditional expression is saved for further backtracking (line 13). Afterwards, the reachability of the true branch is examined and if it is reachable, the true evaluation of the condition will be added to the explored path condition so far (line 17). If the true branch is not reachable the execution will be continued with the false branch by backtracking (line 19).

The `evaluate` function, as depicted in Algorithm 2, takes the conditional expression as input and returns the reachability decision as its output. In this function the decision procedures are tried in order. $DP_{pvi}(.)$ and $DP_{ubox}(.)$ functions refer to the evaluation of the comparison operation in the decision procedures of sections IV and V, respectively. The `propagate_backward` function implements the semantics explained in Section IV-B. The newly evaluated SVIs for each of the operands, R_i and R_j, are propagated backwards with respect to the underlying abstraction. This function refines the possible values of the involving variable in each operand and all expressions in which those were involved until it reaches an input variable. While propagating backwards if the engine detects that further propagation is not possible, for example because of an unsupported operation such as remainder, the function returns a failure. In case neither of the precise and the under-approximating decision procedures can provide a decision, the reachability is decided by the SMT solver (line 18). In

Algorithm 2: `evaluate`

Input : constraint $\mathcal{R}_d = \mathcal{R}_i$ cmp $\mathcal{R}_j$
Output: true or false

1 **if** $\mathcal{A}(\mathcal{R}_i)$ == *pvi* && $\mathcal{A}(\mathcal{R}_j)$ == *pvi* **then**
2   [can_handle, reachability_result] = $DP_{pvi}(\mathcal{R}_d)$;
3   **if** *can_handle == true* **then**
4   **if** *reachability_result == false* **then return** false;
5   r_i = `propagate_backward(`$\mathcal{R}_i$ *when* $\mathcal{R}_d$*, pvi*`)`;
6   r_j = `propagate_backward(`$\mathcal{R}_j$ *when* $\mathcal{R}_d$*, pvi*`)`;
7   **if** r_i == *true* && r_j == *true* **then return** true;
8   **end**
9 **end**
10 **if** $\mathcal{A}(\mathcal{R}_i) \neq$ *bvt* && $\mathcal{A}(\mathcal{R}_j) \neq$ *bvt* **then**
11   [can_handle] = $DP_{ubox}(\mathcal{R}_d)$;
12   **if** *can_handle == true* **then**
13   r_i = `propagate_backward(`$\mathcal{R}_i$ *when* $\mathcal{R}_d$*, ubox*`)`;
14   r_j = `propagate_backward(`$\mathcal{R}_j$ *when* $\mathcal{R}_d$*,ubox*`)`;
15   **if** r_i == *true* && r_j == *true* **then return** true;
16   **end**
17 **end**
18 **if** `check_sat_SMT_solver(`$\mathcal{R}_d$`)` == *true* **then**
19   `upgrade_abstraction(`*bvt*`)`;
20   **return** true;
21 **else**
22   **return** false;
23 **end**

case of a satisfiability decision, the abstraction of the involving symbolic variables are upgraded to bvt and the value set $\mathcal{V}$ of each is updated with the generated witness of the SMT solver using the `upgrade_abstraction` function (line 19).

The default case at line 25 of Algorithm 1 applies the forward propagation of the other operations including addition, subtraction, multiplication, division, and remainder. The `forward_propagation` function applies forward propagation of the current instruction based on the semantics explained in Section IV-A when the abstraction is SVI, or keeps track of symbolic expressions to be used for bit-vector abstraction layer. While symbolically executing the program, the target may be to generate test inputs which trigger an end-point, for example the exit point of the program on a path or an error detection (line 22). Moreover, reaching an end-point triggers a backtracking (line 23). Through the backtracking, the machine restores its previous state for a given pending path. Thus, it undoes the effect of instructions which were executed after a given branching point. This includes undoing trace entries by restoring the previous value for each memory address, pointed by $prev$ element of the trace entry, and updating the pointer to the most recent value of that memory address. Moreover, the path condition is retrieved to the previous state.

Considering the example of Listing 2, a conditional expression is evaluated as follows using `evaluate` function. Variables x and y take two symbolic values represented as two SVIs and their abstraction is set to pvi (lines 2 and 3). When the execution reaches the conditional expression at line 4, the reachability of the false evaluation of the branch is tried first by using the precise SVI decision procedure. An exact decision for evaluating $3 * \langle 10, 30, 1\rangle - 1 \geq 45$ can be made by $DP_{pvi}(.)$. Thus, the values for x will then be refined to $\langle 16, 30, 1\rangle$ using `propagate_backward` function with pvi abstraction and `evaluate` returns $true$. The current execution context will be saved for further backtracking, and then the reachability of the true branch is examined. The evaluation of $3 * \langle 10, 30, 1\rangle - 1 < 45$ is done accordingly using $DP_{pvi}(.)$ and the values for x will then be refined to $\langle 10, 15, 1\rangle$ with pvi abstraction and `evaluate` returns $true$. The recently evaluated constraint will be added to the path condition and the execution continues on the true branch.

```
int x, y;                        1
x = <10, 30, 1>;                 2
y = <10, 20, 1>;                 3
if (3 * x - 1 < 45) {            4
  if (x <= y) {                  5
    if (x > 12) {                6
      ...;                       7
    } else { ...; }              8
  } else { ...; }                9
} else { ...; }                  10
```

Listing 2: An example to show how reachability is evaluated using ASE.

At line 5 to decide the reachability of the false branch, $x > y$, first $DP_{pvi}(.)$ is tried since the abstraction used for representing x and y is still pvi. However, since the operands' value intervals overlap each other, the precise backward propagation is not possible (See Section V). Thus, at next step $DP_{ubox}(.)$ is tried to check if it can provide a decision. Using Observation 2 of Section V, a decision can be made by setting $\langle 13, 15, 1\rangle$ and $\langle 10, 12, 1\rangle$ under-approximating SVIs for operands x and y, respectively. Hence, the abstraction for x and y are set to $ubox$ and `evaluate` returns $true$. The execution context will be saved, and next the reachability of the true branch is investigated. The evaluation of $x \leq y$, is done accordingly using Observation 2 of Section V and the values for x and y will then be refined to $\langle 10, 12, 1\rangle$ and $\langle 12, 20, 1\rangle$ using $ubox$ abstraction and `evaluate` returns $true$. The path condition is updated and the execution continues on the true branch. Later, when line 6 is reached, the current abstraction for x is $ubox$ and thus $DP_{ubox}(.)$ is first checked whether it can provide an exact decision over reachability of the false branch, $\langle 10, 12, 1\rangle \leq 12$, based on the operands' SVIs. The satisfiability is decided by considering values $\langle 10, 12, 1\rangle$ for x using $ubox$ abstraction. The current execution context for false branch is saved and then the true branch is evaluated. The evaluation of $\langle 10, 12, 1\rangle > 12$, however cannot be decided using $DP_{ubox}(.)$ since we only keep an under-approximation of possible values and those values do not satisfy the condition. Therefore, the abstraction is upgraded to bvt and a query, including the symbolic constraint guarded by the path condition, will be sent to the SMT solver to be decided. The value sets for x and y will be refined by the generated witness of the solver (e.g., $\mathcal{V}_x = \langle 14, 14, 1\rangle$ and $\mathcal{V}_y = \langle 14, 14, 1\rangle$) and their abstraction is set to bvt. Now that we reached to an end-point, the execution backtracks to traverse the false evaluations of the conditions starting at the saved context for the false branch of line 6. It is worth mentioning that the interval theory proposed in PARTI [13] cannot reason about this example.

VII. Evaluation

In this section, we demonstrate the effectiveness of our proposed ASE engine in speeding up the symbolic analysis. In Section IV-C we discussed the asymptotic time complexity of the proposed SVI decision procedure. In this section the goal is

benchmark	#paths	ASE (O1)		ASE (O2)		PARTI		baseline		reduction in execution time for ASE (%)			
										versus PARTI		versus baseline	
		time (s)	#queries	time (s)	#queries	time (s)	#queries	time (s)	#queries	O1	O2	O1	O2
bubble_sort_1	300	32.86	0	33.50	0	32.49	0	206.39	67350	-1.13	-3.10	84.08	83.77
bubble_sort_3	55445	670.84	637821	588.35	559587	779.11	728766	1207.21	1135634	13.90	24.48	44.43	51.26
bubble_sort_all	40320	244.69	233087	65.14	91796	255.90	234958	249.14	234958	4.38	74.54	1.79	73.85
heap_sort_1	275	3.22	0	3.20	0	3.11	0	7.11	2340	-3.48	-2.68	54.69	55.04
heap_sort_3	39286	93.41	104524	95.68	105620	140.81	162096	437.36	518822	33.66	32.05	78.64	78.12
heap_sort_all	135423	1188.13	960034	982.57	830632	1202.91	964444	1180.73	964444	1.23	18.32	-0.63	16.78
insertion_sort_1	300	26.23	0	26.32	0	25.53	0	204.87	67350	-2.75	-3.07	87.19	87.15
insertion_sort_3	55445	1391.06	1420769	1208.43	1249913	1575.29	1575736	2830.64	2860000	11.69	23.29	50.86	57.31
insertion_sort_all	40320	73.21	80057	5.22	4878	75.33	80638	74.35	80638	2.81	93.08	1.54	92.98
merge_sort_1	300	0.91	0	0.91	0	0.88	0	1.93	896	-3.99	-3.58	52.80	52.98
merge_sort_3	55445	40.14	29460	23.73	14125	45.12	34746	149.79	147400	11.04	47.41	73.20	84.16
merge_sort_all	40320	71.36	78750	58.94	63238	77.01	80638	76.84	80638	7.34	23.47	7.13	23.30
quick_sort_1	300	19.73	0	19.94	0	19.84	0	18.55	598	0.52	-0.55	-6.34	-7.49
quick_sort_3	49675	274.33	205666	212.65	162382	304.82	230898	558.60	446606	10.00	30.24	50.89	61.93
quick_sort_all	40320	109.53	78800	15.36	9757	111.27	80638	109.93	80638	1.56	86.20	0.37	86.03
selection_sort_1	300	16.60	0	16.49	0	16.10	0	2697.72	1192300	-3.11	-2.43	99.38	99.39
selection_sort_3	60535	3288.82	2913602	2658.36	2480695	3411.95	3030568	4358.47	3828538	3.61	22.09	24.54	39.01
selection_sort_all	101773	623.13	521344	403.23	351101	629.13	526350	625.52	526350	0.95	35.91	0.38	35.54
dijkstra	11632	775.70	88726	724.14	83034	790.76	89990	862.19	99564	1.90	8.42	10.03	16.01
kruskal	7129	746.51	711343	678.89	658255	794.07	738072	830.87	789714	5.99	14.51	10.15	18.29
bellman-ford	326	760.89	91752	766.24	91752	763.01	91752	836.81	102876	0.28	-0.42	9.07	8.43
binary_search_all	4001	0.12	0	0.11	0	904.50	8000	905.90	8000	99.99	99.99	99.99	99.99
linear_find_all	1001	0.04	0	0.04	0	101.71	2000	102.66	2000	99.96	99.96	99.96	99.96
is_permutation	96499	1483.62	1622733	532.04	794099	1649.28	1716634	1620.05	1716634	10.04	67.74	8.42	67.16
gcd	401	272.69	3183	269.13	3183	266.32	3188	279.13	3192	-2.39	-1.05	2.31	3.58
loop_invgen	4098	401.34	22859	262.02	17369	580.93	29846	775.26	34440	30.91	54.90	48.23	66.20
min_max_all	19683	128.90	72067	129.89	74362	139.27	77706	137.76	77706	7.45	6.74	6.43	5.71
dirname	65535	2.05	0	2.06	0	2.01	0	326.42	815764	-1.97	-2.15	99.37	99.37
fibonacci	21	0.20	0	0.19	0	0.19	0	32.68	206554	-2.83	-1.92	99.40	99.41
half	1002	0.03	0	0.03	0	81.73	8006	81.36	8010	99.96	99.96	99.96	99.96
outer_product	15625	0.36	0	0.36	0	189.72	140616	186.51	140616	99.81	99.81	99.81	99.81

TABLE III: The number of explored paths, the symbolic execution time, the number of reached queries to the SMT solver for each of the ASE, the PARTI, and the baseline approaches. O1 and O2 stand for Observation 1 and Observation 2 which are applied in the ASE approach. The reduction in symbolic execution time for the ASE compared to the PARTI and the baseline approaches is reported in percentage and illustrated in last four columns.

to show the efficiency of employing the proposed SVI decision procedure in practice by running the ASE engine on a set of benchmark programs. We evaluate the engine by comparing the following approaches: 1,2) **ASE (O1)** and **ASE (O2)** which refer to the proposed engine and its decision procedure in this paper by applying the heuristic methods of Observation 1 and Observation 2 proposed in Section V, respectively. 3) **PARTI** refers to the state-of-the-art approach [13]. 4) **baseline** is a design strategy which uses an SMT solver for the constraint solving component of the symbolic execution. In baseline all the generated symbolic constraints out of the benchmark programs are sent to the SMT solver as query to be solved.

A. Experimental Setup

We ran our experiments on a 512GB NUMA machine with four 16-core 2.3 GHz AMD Opteron 6376 processors and Linux kernel version 4.15. For our experiments, we used Boolector [25] version 3.2.1 with CaDiCaL as backend SAT solver, and incremental mode enabled. We used GCC and G++ version 9.3 to build Boolector SMT solver and compile our implemented ASE engine source code. We set *depth first search* (DFS) strategy to explore the paths in benchmark programs. The witnesses for each explored path are computed but they are not printed for none of the approaches. While executing, depending on the underlying abstraction the engine keeps the set of values $\mathcal{V}$ (precise or under-approximation) for variables on a path updated for all the approaches.

We run our experiments on a set of benchmarks from different classes of programs typically used in evaluating symbolic execution approaches [13], [26], [27], including sorting, searching, graph, and computational algorithms. The source code of the benchmark programs, which is written in a subset of C, are compiled to a subset of RISC-V using a non-optimizing compiler [28]–[30].

B. Experimental Results

Table III reports the execution results of running all approaches on a set of benchmarks. The reported data includes the number of explored paths (which are identical for all approaches), the symbolic execution time, and the number of queries that reached the SMT solver. The suffixes added to the names of the benchmarks indicate the number of involved symbolic values in their input data (e.g. array, graph), namely *all*, *3*, and *1*. For example, in *bubble_sort_1* and *bubble_sort_3* the input arrays for sorting contain one and three symbolic values respectively and the rest are concrete values. Similarly all the values in the input array for *bubble_sort_all* are symbolic.

The last four columns of Table III report the reduction in symbolic execution time caused by employing ASE using two different heuristic methods of Observations 1 and 2

explained in Section V. Compared to the PARTI and the baseline approaches, on average the ASE engine which is based on Observation 1 led to 17.33% and 45.1% reduction in symbolic execution time, respectively. The application of the ASE engine based on Observation 2 showed an average of 33.62% reduction in symbolic execution time compared to the PARTI approach and an average of 59.84% reduction compared to the baseline. The results show that the ASE engine using either of the heuristic methods of Observations 1 and 2 performs effective while, whenever it does not work, the overhead remains low. Moreover, the usage of the heuristic of Observation 2 for generating under-approximating value intervals often helps deciding more queries and consequently leads to better performance than the heuristic of Observation 1.

The array sorting benchmark programs provide a useful case study for our evaluation. By inserting different number of symbolic values inside the input array to be sorted we can examine the effectiveness of the techniques used in our engine. The size of input arrays for benchmarks with *1*, *3*, and *all* suffixes are 300, 40, and 8, respectively. The inserted symbolic values are represented by value intervals $\langle 0, 2*size, 1\rangle$ where $size$ is the array length. Selecting an interval with a larger upper bound such as $\langle 0, MAX, 1\rangle$ does not influence the number of explored paths for none of the approaches. However, selection of the larger interval $\langle 0, MAX, 1\rangle$ on average causes the under-approximating decision procedure to perform more effectively. As it is shown in Table III for the array sorting benchmarks ASE performs almost as good as the PARTI approach when the number of symbolic values in the input array is one. However, when this number is increased ASE outperforms the PARTI approach. When the number of involving symbolic values is limited to one, the precise SVI decision procedure is enough to make decisions over the set of generated constraints. For sorting algorithms this means that always one side of the involving comparison expressions on the path condition is evaluated to a concrete value. However, when this number is increased, the precise SVI decision procedure may not be able to decide all sets of constraints generated out of the program. In particular when the analysis involves the comparison of variables with overlapping symbolic values (e.g., $x < y$ where $x = \langle 0, 80, 1\rangle$ and $y = \langle 0, 80, 1\rangle$). The interval theory in PARTI does not support the backward propagation of such constraints. Whereas, ASE tries to continue the analysis using the SVI abstraction with under-approximation (Section V), and thereafter if the engine still could not provide a decision using the under-approximation abstraction it sends the set of constraints to the SMT solver. As depicted in Table III the application of the SVI abstraction with under-approximation performs effective for the array sorting benchmarks when more than one symbolic value in the input is involved and it reduces the number of sent queries to the SMT solver.

The results for *bellman-ford* and *gcd* programs show cases that the application of the under-approximating SVI decision procedure does not perform effective. However, the induced overhead is very low which is the result of applying lightweight heuristics to compute under-approximating boxes, and therefore this causes the ASE engine to work almost as good as PARTI. On the other hand, the under-approximating SVI decision procedure performs effective to analyze *binary_search_all* and *linear_find_all* programs for which all the required decisions are made by the precise and under-approximating SVI decision procedures.

As depicted, all the generated constraints out of *dirname*, *fibonacci* benchmarks can be solved by the precise SVI decision procedure. Thus, ASE performs almost as good as PARTI. The *half* and *outer_product* benchmark programs contain expressions with linear symbolic multiplication operation. As explained in Section IV this kind of expression has to be dealt with the value interval abstraction with step. Hence, the interval abstraction proposed in PARTI which only considers value bounds cannot reason about such expressions. This is the reason why ASE outperforms PARTI for these benchmarks.

In summary, we learned that many of the constraints generated out of the programs use a handful of operations with special features that can be solved by the SVI decision procedure. Therefore, employing our theory integration in a layered manner of increasing capability and complexity for constraint solving of symbolic execution is effective. However, a key design factor is to keep the former layers as lightweight as possible so that it does not harm efficiency and performance compared to SMTs or when former layers fail.

VIII. Conclusion

We presented abstract symbolic execution (ASE) with the goal of speeding up reachability decision making in symbolic execution. The ASE engine benefits from a theory integration inside the constraint solving component of symbolic execution in a layered manner of increasing capability and complexity by employing SVI and bit-vector abstractions. We learned through experiments on a set of benchmark programs that many of the constraints generated out of the programs can be solved by the less complex SVI decision procedure. Therefore, employing our theory integration for constraint solving is often effective in reducing symbolic execution time and the number of queries reaching the SMT solver. However, keeping the former layers lightweight is a key design factor to preserve the efficiency of the analysis compared to SMTs or when former layers fail.

Extending the ASE engine to make decisions over floating-point arithmetic is a promising future direction. Our conjecture is that employing the existing, more complicated propagation techniques for the box abstraction leads to an even better speedup for floating-point numbers compared to integers.

Acknowledgment

We thank the ASE reviewers for their thoughtful comments. This work has been supported by the Czech Ministry of Education, Youth and Sports from the Czech Operational Programme Research, Development, and Education, under grant agreement No. CZ.02.1.01/0.0/0.0/15_003/0000421, and the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme, under grant agreement No. 695412.

REFERENCES

[1] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys*, vol. 51, no. 3, pp. 1–39, 2018.

[2] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008, pp. 209–224.

[3] B. Korel, "A dynamic approach of test data generation," in *Proceedings. Conference on Software Maintenance*, 1990, pp. 311–317.

[4] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[5] D. Kroening and O. Strichman, *Decision Procedures, An Algorithmic Point of View*. Springer Berlin Heidelberg, 2008.

[6] M. Brain, D. Kroening, and R. McCleeary, "Algebraic techniques in software verification: Challenges and opportunities," in *Proceedings of the 1st Workshop on Satisfiability Checking and Symbolic Computation*, 2016, pp. 8–12.

[7] S. Bardin, N. Bjørner, and C. Cadar, "Bringing cp, sat and smt together: Next challenges in constraint solving (dagstuhl seminar 19062)," in *Dagstuhl Reports*, vol. 9, no. 2. Schloss Dagstuhl Leibniz Zentrum fuer Informatik, 2019.

[8] R. Sen and Y. N. Srikant, "Executable analysis using abstract interpretation with circular linear progressions," in *Proceedings of the 5th ACM/IEEE International Conference on Formal Methods and Models for Codesign*, 2007, pp. 39–48.

[9] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *IEEE Symposium on Security and Privacy*, 2016, pp. 138–157.

[10] T. Reps, G. Balakrishnan, and J. Lim, "Intermediate-representation recovery from low-level code," in *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, 2006, pp. 100–111.

[11] G. Balakrishnan and T. Reps, "Wysinwyx: What you see is not what you execute," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 32, no. 6, pp. 1–84, 2010.

[12] N. Redini, R. Wang, A. Machiry, Y. Shoshitaishvili, G. Vigna, and C. Kruegel, "Bintrimmer: Towards static binary debloating through abstract interpretation," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2019, pp. 482–501.

[13] O. S. Dustmann, K. Wehrle, and C. Cadar, "Parti: A multi-interval theory solver for symbolic execution," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 430–440.

[14] B. Mayoh, E. Tyugu, and J. Penjam, *Constraint Programming*. Springer Berlin Heidelberg, 1994.

[15] G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey, "Interval analysis and machine arithmetic: Why signedness ignorance is bliss," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 37, no. 1, pp. 1–35, 2015.

[16] A. Gotlieb, M. Leconte, and B. Marre, "Constraint solving on modular integers," in *Proceedings of the Ninth International Workshop on Constraint Modelling and Reformulation*, 2010.

[17] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, Z. Hanna, A. Nadel, A. Palti, and R. Sebastiani, "A lazy and layered smt(bv) solver for hard industrial verification problems," in *Proceedings of the 19th International Conference on Computer Aided Verification*, 2007, pp. 547–560.

[18] L. Hadarean, K. Bansa, D. Jovanović, C. Barrett, and C. Tinelli, "A tale of two solvers: Eager and lazy approaches to bit-vectors," in *Proceedings of the 26th International Conference on Computer Aided Verification*, 2014, pp. 680–695.

[19] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovi'c, T. King, A. Reynolds, and C. Tinelli, "Cvc4," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, 2011, pp. 171–177.

[20] S. Anand, C. S. Păsăreanu, and W. Visser, "Jpf–se: A symbolic execution extension to java pathfinder," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2007, pp. 134–138.

[21] D. Beyer, T. A. Henzinger, and G. Théoduloz, "Program analysis with dynamic precision adjustment," in *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 29–38.

[22] S. Anand, C. S. Păsăreanu, and W. Visser, "Symbolic execution with abstraction," *International Journal on Software Tools for Technology Transfer*, vol. 11, no. 1, pp. 53–67, 2009.

[23] J. Geldenhuys, M. D. B, and W. Visser, "Probabilistic symbolic execution," in *International Symposium on Software Testing and Analysis*, 2012, pp. 166–176.

[24] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002, pp. 58–70.

[25] A. Niemetz, M. Preiner, and A. Biere, "Boolector 2.0: System description," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 9, no. 1, pp. 53–58, 2014.

[26] C. Saumya, J. Koo, M. Kulkarni, and S. Bagchi, "Xstressor : Automatic generation of large-scale worst-case test inputs by inferring path conditions," in *Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification*, 2019, pp. 1–12.

[27] SV-COMP, "Software verification competition," https://github.com/sosy-lab/sv-benchmarks, 2020.

[28] C. M. Kirsch, "Selfie and the basics," in *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2017, pp. 198–213.

[29] A. S. Abyaneh and C. M. Kirsch, "You can program what you want but you cannot compute what you want," in *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*, 2018, pp. 1–15.

[30] Selfie, "An educational software system of a tiny self-compiling c compiler, a tiny self-executing risc-v emulator, and a tiny self-hosting risc-v hypervisor," http://selfie.cs.uni-salzburg.at/.