Type and Interval Aware Array Constraint Solving for Symbolic Execution

Ziqi Shuai
College of Computer, State Key Laboratory of High Performance Computing, National University of Defense Technology
Changsha, China
szq@nudt.edu.cn

Zhenbang Chen∗
College of Computer, National University of Defense Technology
Changsha, China
zbchen@nudt.edu.cn

Yufeng Zhang
College of Computer Science and Electronic Engineering, Hunan University
Changsha, China
yufengzhang@nudt.edu.cn

Jun Sun
School of Information Systems,
Singapore Management University
Singapore
junsun@smu.edu.sg

Ji Wang
College of Computer, State Key Laboratory of High Performance Computing, National University of Defense Technology
Changsha, China
wj@nudt.edu.cn

ABSTRACT
Array constraints are prevalent in analyzing a program with symbolic execution. Solving array constraints is challenging due to the complexity of the precise encoding for arrays. In this work, we propose to synergize symbolic execution and array constraint solving. Our method addresses the difficulties in solving array constraints with novel ideas. First, we propose a lightweight method for pre-checking the unsatisfiability of array constraints based on integer linear programming. Second, observing that encoding arrays at the byte-level introduces many redundant axioms that reduce the effectiveness of constraint solving, we propose type and interval aware axiom generation. Note that the type information of array variables is inferred by symbolic execution, whereas interval information is calculated through the above pre-checking step. We have implemented our methods based on KLEE and its underlying constraint solver STP and conducted large-scale experiments on 75 real-world programs. The experimental results show that our method effectively improves the efficiency of symbolic execution. Our method solves 182.56% more constraints and explores 277.56% more paths on average under the same time threshold.

CCS CONCEPTS
• Software and its engineering → Software testing and debugging.

∗Ziqi Shuai and Zhenbang Chen contributed equally to this work and are co-first authors. Zhenbang Chen and Ji Wang are the corresponding authors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSSTA ’21, July 11–17, 2021, Virtual, Denmark
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8459-9/21/07...
https://doi.org/10.1145/3460319.3464826

1 INTRODUCTION
Symbolic execution [7, 24] provides a way for systematically exploring the space of program paths. Since it was put forward, symbolic execution has been successfully applied in many software engineering activities, including automatic software testing [14, 34], bug finding [20], program repair [17], etc. The success of symbolic execution is built upon the remarkable advancements of constraint solving [18, 25]. At the same time, the effectiveness of constraint solving techniques is also a limiting factor for the success of symbolic execution [7]. First, constraint solving dominates a large part of the time for symbolic execution [14]; second, the program’s complex features need more advanced SMT theories for encoding and solving, such as array or string data types and operations [4, 31]. Therefore, the advancement of constraint solving can improve both the efficiency and effectiveness of symbolic execution.

Array is one of the most basic data types in programming and is widely used in programs. To precisely represent the array operations in the program, many symbolic executors employ the SMT theory for arrays [25], which provides two natural terms (i.e., array Read and Write) for encoding array operations. Usually, the symbolic executor uses the SMT solver combining array theory with other theories (e.g., bit-vector arithmetic theory) for constraint solving. Counter-example-guided abstraction/refinement (CEGAR) based solving method [19] is the state-of-the-art method for array constraint solving, which abstracts the array constraint by eliminating the array terms and refines the abstract constraint by gradually
introducing the axioms defined by the array theory, aiming to find the solution or disprove the constraint faster. Many solvers [12, 19] employed by mainstream symbolic executors [14] implement the CEGAR-based solving method.

However, existing CEGAR-based array constraint solving still suffers from the scalability problems due to the scale of the constraint. We have the following two key observations of array constraint solving in symbolic execution. First, in many cases, the unsatisfiability of the constraint can be decided by a lightweight method, which avoids expensive calls to the underlying SMT solver of the symbolic executor. Second, during the refinement of the CEGAR-based method, there may exist many redundant axioms, particularly for the symbolic execution that models every program’s data as a byte-sized array, which is commonly adopted by the state-of-the-art symbolic executors for precise memory modeling [9].

Based on the above observations, we propose a method for improving array constraint solving in symbolic execution by synergizing constraint solving and symbolic execution. In principle, we use the type and interval information calculated during symbolic execution to boost the array constraint solving. Specifically, we use the type and interval information calculated during symbolic execution and symbolic execution. In principle, we use the type and interval information calculated during symbolic execution to boost the array constraint solving. Specifically, we use the type and interval information calculated during symbolic execution.

We have implemented our optimizations on a state-of-the-art symbolic executor and a CEGAR-based high performance solver, i.e., KLEE [14] and STP [19]. Extensive experimental results on Coreutils benchmark programs and 13 other real-world programs show that our method significantly improves the efficiency of symbolic execution.

The main contributions of this paper are as follows.

- An ILP-based unsatisfiability pre-checking method that can prove the constraint’s unsatisfiability and calculate the intervals of index variables.
- An array access type-guided and index interval-guided optimization method for CEGAR-based constraint solving that removes the redundant axioms in array constraint solving.
- We have implemented the optimizations on the mainstream symbolic executor KLEE and its underlying CEGAR-based solver STP.
- We have carried out extensive experiments on real-world C programs. The experimental results indicate that our optimizations can improve the number of solved constraints and the number of explored paths by 182.56% and 277.56%, respectively.

The remainder of this paper is organized as follows. Section 2 briefly introduces the CEGAR-based constraint solving and illustrates our method by an example program. Section 3 describes the symbolic execution framework and type inference in our method. Section 4 presents our optimizations in detail. Section 5 presents the implementation and the evaluation. Section 6 reviews and compares the related work. Finally, Section 7 concludes.

2 ILLUSTRATION

In this section, we first briefly introduce the state-of-the-art CEGAR-based array constraint solving method. Then, we use an example to illustrate how our approach works.

2.1 CEGAR-Based Array Constraint Solving

An array SMT constraint [25] is a quantifier-free first order logic formula with the following two special functions, where \(a\) is an array variable, \(i\) and \(v\) are index and value variables, respectively.

\[
\mathcal{R}(a, i) \lor W(a, i, v)
\]

(1)

\[
\mathcal{R}(a, i) \rightarrow \mathcal{R}(a, j)
\]

(2)

\[
\mathcal{R}(W(a, j, v), i) = \begin{cases} v & i = j \\ \mathcal{R}(a, i) & \text{otherwise} \end{cases}
\]

(3)

The first one states that two reads must be equal if the index variables are equal. The second one, called read-over-write axiom, states that the value of \(a\)'s \(i\)th element should be modified to \(v\) by \(W(a, j, v)\) and the values of the elements with a different index should remain the same as before. Usually, the array SMT theory is used together with other SMT theories for encoding programs.

Given an array constraint \(C\), a CEGAR-based solving method [19] first eliminates all the write terms in \(C\) by the axiom (3), i.e., using the ITE (If-Then-Else) disjunctive operator [25]. Then, every read term is replaced by a new variable to get an abstract constraint \(C_a\), in which there is no array term. Therefore, initially, \(C_a\) does not have any read axioms. \(C_a\) is solved by other SMT theories. If \(C_a\) is unsatisfiable, \(C\) is unsatisfiable; otherwise, we get a solution \(S\), which will be validated w.r.t. \(C\). If \(C\) is true under \(S\), we find a solution; otherwise, we refine \(C_a\) by adding the (2) axioms (e.g., \(A_0, \ldots, A_n\)) that are violated by \(S\), i.e., \(C_a \land A_0 \land \ldots \land A_n\). The refined constraint will be solved again, and the iteration continues until finding a solution or disproving \(C\).

For example, suppose that \(C\) is the following constraint,

\[
\mathcal{R}_f(a, i) > 10 \land i \geq 0 \land i \leq 3
\]

where \(a\) is \([0, 0, 0, 11]\), and \(C\) has four read axioms w.r.t. (2).

\[
\bigwedge_{n \in \{0, 1, 2\}} i = n \Rightarrow \mathcal{R}_f(a, i) = 0 \land i = 3 \Rightarrow \mathcal{R}_f(a, i) = 11
\]

To solve this constraint using the CEGAR-based method, first \(C_a\) is constructed, which is \(u > 10 \land i \geq 0 \land i \leq 3\) at the beginning (\(u\) represents \(\mathcal{R}_f(a, i)\)). Suppose that the solving of \(C_a\) gets \(u = 12\)
We observe that this constraint can be disproved without invoking worst case after adding all the four axioms.

Then, when checking the feasibility of path $1 \rightarrow 4 \rightarrow 6$, the symbolic executor generates the following array constraint.

$$i + j > 4 \land R_x(a, i) + R_y(a, j) > 10 \tag{5}$$

We observe that this constraint can be disproved without invoking the underlying SMT solver to solve it.

**UNSAT pre-check.** The first optimization is an ILP-based method to pre-check whether an ABV constraint $C$ is unsatisfiable (UNSAT). We abstract the constraints of the index variables in $C$ and compute the interval of each index variable through an ILP solver; then, we compute the intervals of the array read terms and replace them with new variables with the same intervals; finally, we use ILP solver again to check whether the abstracted constraint is UNSAT. The abstracted constraint’s unsatisfiability implies $C$’s unsatisfiability.

Both the problem of solving an ABV constraint and the problem of integer linear programming are NP-Complete in general. However, we can use many abstractions and simplifications to reduce the complexity of the ILP model constructed in symbolic execution. Besides, the interval information computed by the ILP solver can help solve the ABV constraint, which will be elaborated later. For the example program, when we check the unsatisfiability of constraint (5), we first use an ILP solver to compute the minimum and maximum values of $i$. The equations are as follows.

$$i + j > 4$$

$$0 \leq i \leq 3 \tag{6}$$

$$0 \leq j \leq 3$$

We can get that the minimum and maximum values of $i$ are 2 and 3, respectively, i.e., $i$’s interval is [2, 3]. Similarly, $j$’s interval is the same. Then, we can compute the intervals of $R_x(a, i)$ and $R_y(a, j)$ as $[0, 5]$ with the concrete content of array $a$. To eliminate array read terms in the constraint, we introduce two new variables, i.e., $a_i$ and $a_j$, to replace $R_x(a, i)$ and $R_y(a, j)$, respectively. Finally, we get an abstracted version of the ABV constraint as follows.

$$0 \leq a_i \leq 5$$

$$0 \leq a_j \leq 5$$

$$a_i + a_j > 10 \tag{7}$$

Now, we use ILP to decide the unsatisfiability of the above abstracted constraint. As it is unsatisfiable, the result of solving the constraint (5) is UNSAT, and we do not need to invoke the ABV solver.

Let us consider another case: if we replace Line 2 with Line 3 in the example program in Figure 1(a), i.e., the value of the last element in the array $a$ is 9 instead of 5, the above UNSAT pre-checking method cannot prove the unsatisfiability of the constraints (5), which is satisfiable. However, the ABV constraint’s satisfiability is still unknown as the answer is for the abstracted constraint. So we must invoke the underlying SMT constraint solver. This constraint is encoded as follows in the underlying SMT solver that employs byte-level reasoning, where $\circ$ denotes the bit concatenation operator.
Axiom elimination. Inspired by the idea of typed memory modeling [8, 13], we observe that many axioms are redundant if we have type information of the array accesses and interval computation of the program’s index variables. For example, for the array constraint (8), because every array access in the program is reading an integer, there is no need to have an axiom for the two bytes that have different offsets in the integers. For the axiom (9) is redundant because \( i_0 \) is the first byte in the integer, and the 15th byte is the fourth byte; whereas the axiom (10) is necessary because both the \( i_1 \)th byte and the \( i_2 \)th byte are the second byte. Besides, because we have computed each index variable’s interval in the UNSAT pre-checking step, there is no need to have an axiom for any byte within the interval and any byte outside of the interval. For example, the below axiom is redundant because \( i_0 \)’s interval is [8, 12].

\[
i_0 = 0 \Rightarrow R_{kb}(a, i_0) = 0
\]

In the second and third parts of Figure (1b), the axioms represented by gray lines are redundant. In this way, we can reduce the number of axioms from 156 to 20 (i.e., 2 * 8 + 4), which significantly reduces the complexity of CEGAR-based array constraint solving.

3 SYMBOLIC EXECUTION FRAMEWORK

In this section, we first briefly introduce the symbolic execution framework. Then, we present the rules for the type inference in symbolic execution.

\[
P ::= \text{var } a[ e ] : T \mid a := e \quad P \triangleright P \mid \text{if } e \text{ P } \mid \text{while } e \text{ P }
\]

\[
e ::= c \mid a \mid e \oplus e \mid e \cdot (T)*
\]

Figure 2: Syntax of a core language.

3.1 Basic Framework

Let \( T \) be the set of atomic types, \( N \) be the name set, and \( C \) be the set of constants. Without losing of generality, we consider the programs defined by the language in Figure 2 for brevity, where \( T \subseteq T, a \in N, c \in C, \) and \( \oplus \) represents a commonly used boolean, numeric, or bit operator.

Note that in our language, the only variables are array variables, which are pointers. There are three atomic statements: array variable declaration, array variable assignment, and memory content update. Besides, we provide three typical composition operators for composite statements. In the expressions, we provide pointer dereference \( *e \) and pointer type conversion \((T)\cdot e\), which are typical for memory operations. In principle, the language is expressive enough for modeling C-like programs. Our implementation fully supports C programs.

During the symbolic execution of a program \( \mathcal{P} \), a symbolic state is a tuple \((\sigma, M, \mathcal{J})\), such that:

- \( \sigma = (\Delta, H, stmt, PC) \), where \( \Delta \) is the variable map that maps each array variable to its address range, \( H \) is the heap map that maps an address to its concrete or symbolic byte value, \( stmt \) is the next statement to be executed, and \( PC \) is the current path constraint, i.e., an ABV constraint. We use \( *e \) to denote the element \( e \) of \( \sigma \), e.g., \( *astmt \) is \( \sigma \)'s statement to be executed.

- \( M \) is the map that gives the size of an array variable’s access type.

- \( \mathcal{J} \) maps an array variable to its address range.

We use \( \sigma(v) \) to denote the address value of array variable \( v \) and \( \sigma(e) \) to denote the value of the expression \( e \) on a symbolic state.

Algorithm 2 shows the symbolic execution algorithm. The algorithm employs a worklist-style procedure. At the beginning, the worklist only contains the initial symbolic state \( s_i \), where \( \sigma = (0, 0, stmt_i, true) \), where \( stmt_i \) is \( \mathcal{P} \)'s entry statement, and \( M \) and \( \mathcal{J} \) are both 0. The algorithm adopts the traditional symbolic execution in the state-forking style [7]. The algorithm selects a state from worklist. It carries out the symbolic execution of the next statement on the state, which updates the state and may generate and insert new states into worklist. The symbolic execution of each statement is standard [24] and omitted for brevity. Along with symbolic execution, the algorithm also infers the type information of each array’s accesses, and the type information \( M \) will be used later to improve the constraint solving. We will explain the type inference rules in the next subsection.

3.2 Type Inference of Array Accesses

Figure 3 and Figure 4 show the inference rules for atomic statements and expressions, where \( S(T) \) represents the size of the type \( T \). For a statement \( s \) or an expression \( e \), denoted by \( a \), we define the following inference relation.

\[
(M, \mathcal{J}) \xrightarrow{\sigma, a} (M', \mathcal{J}')
\]
Type and Interval Aware Array Constraint Solving for Symbolic Execution

Algorithm 1: Symbolic Execution Framework

SE(\mathcal{P})

Data: \mathcal{P} is a program

begin
1. worklist ← \{s_i\}
2. while worklist ≠ \emptyset do
3. \ (\sigma, M, \mathcal{G}) ← Select(worklist)
4. if \sigma.stmt is atomic and not declaration then
5. \ (M, \mathcal{G}) \xrightarrow{\sigma, \sigma.stmt} (M', \mathcal{G})
6. \ M ← M'
7. if \sigma.stmt has a branch condition e then
8. \ (M, \mathcal{G}) \xrightarrow{e} (M', \mathcal{G})
9. \ M ← M'
10. end
11. \ S ← Execute(\sigma, M, \mathcal{G})
12. if \sigma.stmt is declaration then
13. \ for (\sigma_n, M_n, \mathcal{G}_n) ∈ S do
14. \ (M_n, \mathcal{G}_n) \xrightarrow{\sigma_n, \sigma.stmt} (M', \mathcal{G}')
15. \ end
16. end
end

Figure 3: Type inference rules for atomic statements.

It means that the type map \ M and the array range map \mathcal{G} are updated to \ M' and \mathcal{G}' by a \sigma. Type inference only needs to consider atomic statements and expressions. The key idea is to infer an array’s access type as the minimum type size of the pointer dereferences inside the array’s address range.

When an array variable \ v is declared, we use the \sigma after executing the statement to infer \ v's access type. We record the type of \ v's access as \ T at the beginning and \ v's address range for later inferences (Rule 1 in Figure 3). For the array variable assignment and the memory content update statements, we infer the type information by the expressions based on \sigma before executing the statement. The constant and variable expressions do not change \ M and \mathcal{G}. The most important one is the type conversion expression (Rule 3 in Figure 4), which modifies the array’s access type size to the minimum one of the target type \ T and the current type size. We update \ M and \mathcal{G} by the composed expressions for the deference expressions and binary composite expressions.

Hence, during symbolic execution, if the current statement is atomic and not a variable declaration statement, we infer the type before the statement’s symbolic execution (Lines 6-7). If the statement is a composite statement and has a branch expression, we also infer the type by the expression before symbolic execution (Lines 10-11). For the variable declaration statement, we infer the type after the symbolic execution (Lines 16-17).

4 ARRAY CONSTRAINT SOLVING

This section first introduces the basic framework of our CEGAR-based solving method. Then, we discuss the UNSAT pre-checking method. Next, we explain the axiom generation algorithm.

4.1 CEGAR-Based Framework

Algorithm 2 shows the details of our CEGAR-based solving method for array constraints. The inputs are an ABV constraint \ C and the array accesses’ type information \ M inferred by the symbolic executor. The algorithm returns UNSAT if \ C is unsatisfiable; otherwise, the algorithm returns a solution if \ C is satisfiable.

The algorithm first uses Pre-check (c.f., Algorithm 3) to check whether \ C is unsatisfiable. If Pre-check returns UNSAT, the algorithm returns UNSAT. Otherwise, it records the interval map \ J from Pre-check which maps index variables in \ C to their intervals. Then, the algorithm computes the axioms that will be added during CEGAR refinement iterations based on \ M and \ J (c.f., Algorithm 4). Next, it constructs an abstract constraint \ C_a [19]. The abstraction is as follows.

- Use the read-over-write axiom (c.f., Axiom (3)) to eliminate the write terms in \ C.
- Replace each read term \ R_i with a new fresh bit-vector variable \ v_i and record the mapping between \ R_i and \ v_i.

Therefore, after abstraction, \ C_a is a bit-vector constraint without any array terms. \ C_a is an over-approximation of \ C (i.e., \ C ⇒ C_a), since there are no read axiom requirements in \ C_a. The CEGAR procedure starts with \ C_a.
Algorithm 2: CEGAR-based Solving Framework

CEGAR-ABV($C, M$)

Data: $C$ is an ABV constraint, and $M$ maps $C$’s each array to the size of its access type

1 begin
2 ($r, J$) ← Pre-check($C, M$) → $J$ is an interval map
3 if $r = \text{UNSAT}$ then
4    return UNSAT
5 end
6 $A ←$ Axioms($C, M, J$)
7 $C_a ←$ abstract($C$)
8 while true do
9    ($r, S$) ← sat($C_a$)
10   if $r = \text{UNSAT}$ then
11      return UNSAT
12 end
13 else if $r = \text{SAT}$ then
14    if $S \models C$ then
15      return $S$
16 end
17 else
18    $A_S ←$ Select($S, A, C$)
19    $C_a ← C_a \land A_S$
20    $A ← A \setminus A_S$
21 end
22 end
23 end

In the CEGAR loop, $C_a$ is solved as a bit-vector constraint. The algorithm converts $C_a$ to an SAT problem [25] and employs an SAT solver for solving (Line 9). If the result $r$ is UNSAT, the algorithm returns UNSAT, because $C_a$ is implied by $C$. If $C_a$ is satisfiable, the algorithm checks whether the solution $S$ satisfies $C$, i.e., $S \models C$, which includes all the array axioms in the form (2). If $S$ satisfies $C$, a solution is found; otherwise, the algorithm employs a procedure to select the axioms that are not satisfied by $S$. Then, $C_a$ is refined by the axioms (Line 19) that will be removed from $A$ (Line 20). This iteration continues until a solution is identified, or $C$ is proved to be UNSAT or timeout (omitted for brevity). In this way, the CEGAR-based algorithm tries to find a solution or disprove the constraint by solving a simplified version of the original constraint.

4.2 UNSAT Pre-checking

Algorithm 3 shows the details of the UNSAT pre-checking of the input constraint $C$. The basic idea is to abstract $C$’s constraint of index variables with an integer arithmetic constraint. Then, we employ an ILP solver to compute each index variable’s range of values, based on which we compute the interval of read terms. Next, we introduce new variables to replace read terms and transform the final array-term-free constraint to integer arithmetic format for checking the unsatisfiability of $C$.

To begin with, the algorithm partitions constraint $C$ into a part without array terms $C_V$ and a part with array terms $C_{arr}$. Then, for each array read term $r$, suppose that the index variable of $r$ is $v$ (denoted as $\text{indexVariable}(r)$), we linearize $v$’s related constraints in $C_V$ (denoted as $C_V \downarrow v$) to an integer arithmetic constraint $C_v^a$ in which no disjunction exists (Line 5). Then, if we set $v$ as objective, we can use ILP to compute $v$’s upper and lower bounds, i.e., its range (Lines 6). With the range of $v$ and the type map $M$, we can compute $r$’s interval (Lines 7) [27]. After all intervals of read terms are computed, we linearize $C_{arr}$ to an integer arithmetic constraint $C_{arr}^a$ and use ILP again to check whether it is unsatisfactory (Lines 9-10). The algorithm will return UNSAT if $C_{arr}^a$ is unsatisfiable; otherwise, the algorithm returns UNKNOWN and the interval map, then we can use the interval map to help the CEGAR-based solving later. Note that the algorithm may return UNSAT as well when checking $C_v^a$. Here we omit it for brevity.

The algorithm employs multiple abstractions. First, the algorithm abstracts the constraint $C_V$ of index variables (linearize$_a(C_V \downarrow v)$ at Line 5), and $C_V \downarrow v$ is a bit-vector constraint. We employ the method in [37][11] to abstract a bit-vector constraint to an ILP problem. We consider each variable as an unsigned variable and use unsigned numeric operations to represent signed numeric operations. The overflow behavior of bit-vector variables is modeled. Second, the algorithm abstracts the read terms in the ABV constraint by their intervals and returns to the first abstractions again (linearize$_a(C_{arr}, I_r)$ at Line 9). Third, to simplify the ILP problem, the algorithm adopts several abstraction rules when precise model-checking is costly. These abstraction rules ensure over-approximation. Suppose the bit-vector constraint $C_V \downarrow v$ of an index variable $v$ is as follows, where each $c_i$ is an atomic bit-vector constraint.

\[
\bigwedge_{i=1}^{n} c_i \tag{12}
\]
If there exists symbolic array writes in $C_Y \downarrow v$, the UNSAT pre-checking method is skipped. Otherwise, the abstraction for $C_Y \downarrow v$ (i.e., Algorithm 3’s $\text{linearize}_Y$ at Line 5) does the abstraction for each $c_i$.

The key idea is to exclude the constraints of complex operators or abstract the constraint by introducing a new variable with a larger range, which ensures the original constraint’s over-approximation. We exclude the $c_i$ in which one of the following conditions holds.

- $c_i$ is a comparison constraint, and the comparison operator is the not equal operator.
- $c_i$’s comparison operator is a signed operator and both operands are not constant; or if one of the operands is a constant and the other operand is a variable that could be negative or positive.
- There exists any non-linear expression in $c_i$.

Besides, there are following abstraction rules with respect to $c_i$’s constraint form.

- $c_i$’s each boolean predicate is replaced by a new boolean variable.
- If $c_i$ is a (urem x c) expression, it is abstracted to a variable $v$ with a specific interval of $[0, c]$ when $c$ is a constant.
- If $c_i$ is a (xor a b) expression, it is abstracted to a bit-vector variable $v$ whose bit-width is the same as $a$ and $b$.
- If $c_i$ is a (bvor a b) expression, it is abstracted to a variable $v$ that satisfies $0 \leq v \leq a$ and $0 \leq v \leq b$. However, if the binary of $a$ or $b$ is a sequence of 1 following a sequence of 0, precise modeling is available according to the way that Extract expression is modeling [11].
- If $c_i$ is a (bvor a b) expression, suppose $a$ and $b$ is a $k$-bit bit-vector, the expression is abstracted to a variable $v$ that satisfies $a \leq v \leq 2^k$ and $b \leq v \leq 2^k$.
- $c_i$’s each read of symbolic array is abstracted to a variable $v$ with a specific interval of $[0, 255]$.

In addition, if an index variable’s interval is larger than the array’s range, we add the constraints to require that the variable should be in the array’s range.

Finally, to further reduce the ILP problem’s cost, we propose two kinds of simplifications: interval computation and integer introduced when modeling the modulo semantics [11] of bit-vector operations. If the result of a bit-vector operation does not overflow, there is no need to model the modulo semantics. A typical bit-vector expression in symbolic execution is as follows, where $ZE_{32}$ denotes the 32-bit zero-extend operator.

$$(((ZE_{32} a_{[8]}) \times_{[32]} 2) +_{[32]} 1)$$

Because $a$ is an 8-bit variable, interval computation can check that the bit-wise multiplication and the bit-wise addition do not overflow. The other simplification, i.e., caching, reuses the ILP solutions among the solving of the different constraints in symbolic execution. When the constraints are similar in structure, caching reduces the pre-checking’s overhead a lot.

### 4.3 Axiom Generation

Algorithm 4 shows the axiom generation of an ABV constraint $C$ with the type information $M$ of the array accesses and the interval information $J$ of the array index variables in $C$. For each array $a$ and each $a$’s index variable $v$ with the offset $i$, we first get $v$’s interval from $J$ (Line 5). Then, we generate an axiom for $v$ and the $a$’s element with the same offset (Line 8) and within $v$’s interval. Besides, we generate an axiom for the different index variables with the same offset (Line 15).

As mentioned previously, to get the interval of each index variable, we construct an ILP model for the over-approximation of the variable’s related constraints. As expected, the computed interval may be an over-approximation of the exact interval of the index variable, i.e., the lower bound is smaller, and the upper bound is larger. Therefore, the index variable’s value is certainly not equal to those values outside of the interval, which means that the axioms related to those values can be removed safely. So, the interval-based axiom elimination only removes redundant axioms. The type-based axiom elimination has the same guarantee because the axioms between the bytes with different offsets are also redundant. Furthermore, type inference rules in Section 3.2 prefer to record a smaller access type, which guarantees that the necessary axioms will never be removed. Therefore, the constraint after eliminating redundant axioms is equivalent to the original one.

### 4.4 Discussion

The pre-checking algorithm employs a lightweight procedure to check the abstract constraint’s unsatisfiability, which implies the original constraint’s unsatisfiability. The abstraction’s precision determines the precision of the index variables’ ranges, which directly determines the extent to which the pre-checking can prove
the constraint’s unsatisfiability. In the case that Pre-check fails to conclude UNSAT, the index variables’ intervals can help remove the redundant axioms in the later solving procedure. The pre-checking algorithm is general and can be applied to any array constraint solver, but the abstraction method may differ. In principle, the abstraction needs to tradeoff the precision and the pre-checking’s overhead.

The type and interval aware axiom generation method is enabled by the synergy between symbolic execution and constraint solver. Symbolic execution infers the type information of array accesses, which is used to do pre-checking and remove the redundant axioms in the ABV constraint solving. The type inference rules guarantee the correctness of solving by using the minimum access size of the array. Hence, in the worst case, the access type of an array has the one-byte size, which generates the same axioms as before. However, in practice, the size of an array’s access type is often larger than one. On the other hand, the interval information collected in the pre-check algorithm can be used to remove the redundant axioms. In a word, the type-aware axiom optimization is applicable if the symbolic executor employs byte-sized array-based modeling. The interval-aware axiom optimization applies to any array theory solver despite the usage of the CEGAR-based method.

5 IMPLEMENTATION AND EVALUATION

We have implemented our approach on KLEE [14] and STP [19]. We implemented the type inference rules in KLEE. We have implemented the pre-checking algorithm in KLEE for the first optimization, and the pre-checking will be carried out before invoking STP. We use PPL [6] for integer linear programming, and the abstraction is implemented for the constraint in KLEE’s KQuery language [3]. For the second optimization, axiom generation is implemented in STP, and STP’s interfaces are modified to support type information input and interval information input.

5.1 Research Questions

There are the following three research questions:

- **RQ1**: effectiveness, i.e., can our method improve the efficiency of symbolic execution?
- **RQ2**: relevance of either optimization, i.e., how about each optimization’s significance for improving effectiveness?
- **RQ3**: compare with the dedicated optimization method for array constraint solving, i.e., KLEE-Array [31], how about our method’s effectiveness?

5.2 Experimental Setup

To answer the research questions, we have applied our implementation on the benchmark programs in Table 1 to evaluate the optimization methods. GNU Coreutils\(^2\) is a standard benchmark suite for KLEE-based implementations. Among all the programs in this benchmark (which has a total of 89 programs), we filter those that are irrelevant (i.e., no array constraints that trigger CEGAR-based solving), and the remaining contains 62 programs. LD and BC are two GNU programs, which are used in [31] as the benchmark for the optimizations in symbolic execution for arrays. APR is used in [23] as the benchmark for a new memory model for symbolic execution. Other programs are lexers or scanners of different grammar. In total, we have collected 75 real-world open-source C programs.

In principle, the number of solved constraints and explored paths directly reflect a symbolic executor’s efficiency. If a symbolic executor can solve more constraints or explore more paths under a time threshold, it is considered as more efficient. Hence, we first use KLEE to analyze the benchmark programs under different configurations and then collect the solved constraints and the explored paths during symbolic execution. To alleviate the randomness of the experiments, we use the depth-first search (DFS) search heuristic. We analyze each program in 30 minutes. Finally, we carried out all the experiments three times on a server whose CPU is 3.1GHz and got the average values. The operating system is Ubuntu 16.04.

5.3 Experimental Results

Answer to RQ1. Figure 5 shows the results of constraint solving using our optimizations without KLEE’s optimizations in analyzing the 75 programs. The Y-axis shows the relative increase of

![Figure 5: Result of queries without the query optimizations in KLEE under DFS.](image-url)
with KLEE’s optimizations, we can increase the number of queries which illustrates the effectiveness of the caching simplification in where the number of queries is decreased. As shown by the figure, the queries, which is calculated as follows. \( Q_{\text{opt}} \) is the number of queries (i.e., constraints) solved under our optimizations, and \( Q_{\text{ori}} \) is the one without our optimizations.

\[
\frac{Q_{\text{opt}} - Q_{\text{ori}}}{Q_{\text{ori}}} \tag{13}
\]

The X-axis shows the program numbers. The relative increasing values order the programs, and the 29th program is the last program number where the number of queries is decreased. As shown by the figure, our method can increase the number of solved constraints (often significantly) for 46 programs (61.33%) and decrease the number (always only slightly) for 29 programs. Since ILP solving introduces overhead, it is understandable that performance degradation occurs in some cases. However, as shown in Figure 5, degradation is minor, which illustrates the effectiveness of the caching simplification in ILP solving. On average, the relative increasing value for queries is 160.52% (−15.63%−2335.80%), and the value for explored paths is 80.04% (−55.31%−1206.63%).

To further evaluate the compatibility of our method and KLEE’s optimizations, we also run the experiment with all query optimizations in KLEE. Same as Figure 5, Figure 6 shows the query results under our optimizations with all of KLEE’s query optimizations. The 6th program is the last program, where the number of queries is decreased. There are 13 programs on which our optimizations have no effect because KLEE’s optimizations are efficient enough to reduce the underlying constraint solver’s most invocations. Hence, with KLEE’s optimizations, we can increase the number of queries for 56 programs (74.67%). The average increase in the number of queries is 182.56% (−0.56%−2271.43%), and the value for explored paths is 277.56% (−1.63%−10824.39%).

To further evaluate our method, we select the benchmark programs whose ratio of the queries that reach the CEGAR-loop in the solver is greater than 10% (with respect to the results in Figure 5) for a further evaluation under different configurations, which we believe is a relatively appropriate ratio. There are 23 programs. We compare four configurations in detail with KLEE’s optimizations: vanilla KLEE with query optimizations\(^3\), using assertion encoding\(^4\), using the pre-check method (Opt 1) and using both optimizations (Opt 1+2).

Table 2 provides the detailed results of these 23 programs. The first column displays the program names and time spent in the constraint solver under vanilla KLEE. The second column shows two metrics: solved queries and explored paths. Then, the remaining columns show the results of different configurations. For the last four configurations, the table also shows the relative increasing values of solved queries and explored paths. For the sake of the reader’s convenience, we highlight the maximum value in each case under the five configurations. It is then easy to find out which configuration performs best with respect to different metrics in each case.

The assertion encoding achieves the average relative increasing of queries and paths as 320.19% (−0.95%−1410.99%) and 324.78% (−17.24%−3936.59%) respectively. Our first optimization achieves the average relative increasing of queries and paths as 339.78% (0%−1474.73%) and 351.50% (−17.24%−4231.71%) respectively. If both optimizations are used, the achieved average relative increasing of queries and paths are 592.89% (0%−2271.43%) and 896.98% (3.44%−10824.39%), respectively. Our optimizations solved more queries and explored more paths in all 23 programs than vanilla KLEE. Hence, our optimizations are effective. Figure 7 shows the results of the 13 programs whose time of constraint solving dominates the total time of symbolic execution (more than 80%). As shown by the figure, employing both optimizations improves the results for all the 13 programs, and the average relative increases of queries and paths are 1046.88% (348.85%−2271.43%) and 1579.10% (26.41%−10824.39%), respectively. These results indicate that our method is more effective for the programs whose symbolic execution is solving-intensive.

**Statement coverage.** In the following, we show an end-to-end improvement of our method for symbolic execution’s application
Table 2: Solved queries and explored paths results of the 23 programs whose ratio of the constraints that enters the CEGAR-based solving loop is at least 10%. The programs are ordered by the ratio of solving time in symbolic execution.

<table>
<thead>
<tr>
<th>Program (ST%)</th>
<th>Metrics</th>
<th>Vanilla</th>
<th>Assertion (Inc%)</th>
<th>Opt 1 (Inc%)</th>
<th>Opt 1+2 (Inc%)</th>
<th>KLEE-Array (Inc%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>apr (99.4%)</td>
<td>Queries</td>
<td>347</td>
<td>883(154.4%)</td>
<td>986(184.15%)</td>
<td>4940(1323.63%)</td>
<td>4414(1172.05%)</td>
</tr>
<tr>
<td></td>
<td>Paths</td>
<td>100</td>
<td>1664(164.04%)</td>
<td>1734(164.04%)</td>
<td>5542(5442.0%)</td>
<td>3456(3565.0%)</td>
</tr>
<tr>
<td>ld (84.02%)</td>
<td>Queries</td>
<td>1046</td>
<td>440(320.75%)</td>
<td>4856(364.24%)</td>
<td>5641(439.29%)</td>
<td>1857(77.53%)</td>
</tr>
<tr>
<td></td>
<td>Paths</td>
<td>462</td>
<td>524(13.42%)</td>
<td>525(13.43%)</td>
<td>584(26.41%)</td>
<td>489(5.84%)</td>
</tr>
<tr>
<td>mkfifo (34.99%)</td>
<td>Queries</td>
<td>808</td>
<td>803(6.62%)</td>
<td>810(0.25%)</td>
<td>819(1.36%)</td>
<td>638(21.04%)</td>
</tr>
<tr>
<td></td>
<td>Paths</td>
<td>148588</td>
<td>146508(1.4%)</td>
<td>150975(1.61%)</td>
<td>158096(6.4%)</td>
<td>170580(14.8%)</td>
</tr>
<tr>
<td>tac (30.48%)</td>
<td>Queries</td>
<td>1063</td>
<td>107(15.31%)</td>
<td>1080(1.41%)</td>
<td>1099(3.19%)</td>
<td>744(30.42%)</td>
</tr>
<tr>
<td></td>
<td>Paths</td>
<td>181133</td>
<td>1890(14.35%)</td>
<td>189391(4.56%)</td>
<td>199560(10.17%)</td>
<td>186594(3.01%)</td>
</tr>
<tr>
<td>head (18.19%)</td>
<td>Queries</td>
<td>1122</td>
<td>1133(9.98%)</td>
<td>1237(10.25%)</td>
<td>1125(0.27%)</td>
<td>681(39.39%)</td>
</tr>
<tr>
<td></td>
<td>Paths</td>
<td>225386</td>
<td>229105(1.65%)</td>
<td>204987(9.05%)</td>
<td>303754(34.77%)</td>
<td>227922(11.13%)</td>
</tr>
<tr>
<td>touch (15.70%)</td>
<td>Queries</td>
<td>1125</td>
<td>1147(9.16%)</td>
<td>1149(2.13%)</td>
<td>1179(4.84%)</td>
<td>631(43.91%)</td>
</tr>
<tr>
<td></td>
<td>Paths</td>
<td>2109590</td>
<td>219859(4.4%)</td>
<td>220742(4.82%)</td>
<td>233603(10.93%)</td>
<td>218317(6.67%)</td>
</tr>
<tr>
<td>mkdir (15.20%)</td>
<td>Queries</td>
<td>733</td>
<td>732(4.96%)</td>
<td>738(0.68%)</td>
<td>744(1.5%)</td>
<td>530(27.69%)</td>
</tr>
<tr>
<td></td>
<td>Paths</td>
<td>176282</td>
<td>169954(3.39%)</td>
<td>180342(2.3%)</td>
<td>185129(9.02%)</td>
<td>201180(14.12%)</td>
</tr>
<tr>
<td>du (12.89%)</td>
<td>Queries</td>
<td>846</td>
<td>853(8.03%)</td>
<td>857(1.3%)</td>
<td>868(2.6%)</td>
<td>630(25.53%)</td>
</tr>
<tr>
<td></td>
<td>Paths</td>
<td>1635273</td>
<td>170106(2.92%)</td>
<td>173134(4.76%)</td>
<td>181855(10.03%)</td>
<td>168268(8.18%)</td>
</tr>
<tr>
<td>unexpand (12.18%)</td>
<td>Queries</td>
<td>189</td>
<td>189(0.0%)</td>
<td>189(0.0%)</td>
<td>189(0.0%)</td>
<td>189(0.0%)</td>
</tr>
<tr>
<td></td>
<td>Paths</td>
<td>862561</td>
<td>875051(1.45%)</td>
<td>885281(2.7%)</td>
<td>913214(5.87%)</td>
<td>84969(1.49%)</td>
</tr>
<tr>
<td>kill (11.52%)</td>
<td>Queries</td>
<td>2564</td>
<td>2616(2.03%)</td>
<td>2718(6.01%)</td>
<td>2820(9.98%)</td>
<td>2645(5.16%)</td>
</tr>
<tr>
<td></td>
<td>Paths</td>
<td>286276</td>
<td>295502(3.22%)</td>
<td>306636(7.11%)</td>
<td>317902(11.05%)</td>
<td>298256(4.18%)</td>
</tr>
<tr>
<td>wc (6.34%)</td>
<td>Queries</td>
<td>164</td>
<td>164(0.0%)</td>
<td>164(0.0%)</td>
<td>164(0.0%)</td>
<td>164(0.0%)</td>
</tr>
<tr>
<td></td>
<td>Paths</td>
<td>606685</td>
<td>605447(4.00%)</td>
<td>615035(1.54%)</td>
<td>634648(7.87%)</td>
<td>591048(2.42%)</td>
</tr>
<tr>
<td>pr (5.34%)</td>
<td>Queries</td>
<td>351</td>
<td>363(3.42%)</td>
<td>363(3.42%)</td>
<td>363(3.42%)</td>
<td>349(5.57%)</td>
</tr>
<tr>
<td></td>
<td>Paths</td>
<td>111046</td>
<td>106501(4.0%)</td>
<td>112157(1.0%)</td>
<td>114868(3.44%)</td>
<td>119509(7.62%)</td>
</tr>
</tbody>
</table>

to some software engineering activities. We have applied KLEE equipped with the optimized STP to analyze the thirteen large or complex benchmark programs (i.e., the ones besides Coreutils programs) in Table 1 in five hours. Figure 8 shows the coverage results of 13 programs. As shown in the figure, our optimizations can improve the statement coverage for 12 programs except for apr. For apr, our optimizations improve the number of instructions to 5 times; whereas, it does not contribute to statement coverage.

These results indicate that the advancement in constraint solving can directly benefit symbolic execution's applications. In summary, the answer to the first research questions is as follows.

**Answer to RQ1:** Our optimizations can effectively improve the efficiency of symbolic execution. On average, the optimizations can increase the relative queries by 182.56% and the relative
Table 3: Results of explored paths and executed instructions (vs. KLEE-Array)

<table>
<thead>
<tr>
<th>Programs</th>
<th>KLEE-Array</th>
<th>Our Method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#Instrs</td>
<td>#Paths</td>
</tr>
<tr>
<td>yaml</td>
<td>71667</td>
<td>29</td>
</tr>
<tr>
<td>rust</td>
<td>38892</td>
<td>24</td>
</tr>
<tr>
<td>sgml_lex</td>
<td>599397</td>
<td>184</td>
</tr>
<tr>
<td>clan</td>
<td>69777</td>
<td>66</td>
</tr>
<tr>
<td>rats_py</td>
<td>353230</td>
<td>342</td>
</tr>
<tr>
<td>clex</td>
<td>87322</td>
<td>37</td>
</tr>
<tr>
<td>libguile</td>
<td>35871</td>
<td>22</td>
</tr>
<tr>
<td>rats_php</td>
<td>5221268</td>
<td>1554</td>
</tr>
<tr>
<td>apr</td>
<td>637629</td>
<td>3456</td>
</tr>
<tr>
<td>bc</td>
<td>340874</td>
<td>36</td>
</tr>
<tr>
<td>rats_perl</td>
<td>325398</td>
<td>338</td>
</tr>
<tr>
<td>libguile</td>
<td>665723</td>
<td>337</td>
</tr>
<tr>
<td>ld</td>
<td>373181619</td>
<td>489</td>
</tr>
</tbody>
</table>

**Figure 8:** Result of statement coverage. The gray part is the statement coverage of vanilla KLEE with query optimizations. The darker part is the improvement of our method.

paths by 277.56%. Besides, the optimizations can increase the statement coverage for large-scale real-world programs.

**Answer to RQ2.** In the following, we report experimental results on the relevance of either optimization. We observe that the second optimization makes a noticeable improvement in terms of the number of solved queries. We further observe that the first optimization itself does not contribute much to increase the number of solved queries. This is expected as the first optimization only allows to conclude UNSAT early. However, the first optimization contributes significantly to the effectiveness of the second optimizations.

For the 23 programs, there are 19 cases (82.61%) in which the first optimization can improve the solved queries compared with assertion encoding. As for explored paths, the number is 16. For the 23 cases, the average improvement of solved queries is 7.18% (~5.82%~18.76%); the average improvement of explored paths is 5.93% (~10.53%~30.59%). Hence, although the first optimization can improve constraint solving efficiency in many cases, it may not increase the explored paths as the behavior of KLEE’s caching optimizations depends on the solving results. On the other hand, if we employ the second optimization, compared with assertion encoding, we can achieve 92.09% (0.71%~459.46%) improvement for solved queries and 99.48% (4.36%~233.05%) improvement for explored paths, respectively.

**Answer to RQ2:** The second optimization is more significant than the first one. The first optimization can generate useful information to help the second optimization.

**Answer to RQ3.** KLEE-Array [31] is the state-of-the-art work for optimizing array constraint solving in symbolic execution. KLEE-Array optimizes the array constraints by simplifying and transforming array constraints into the constraints without arrays before invoking the solver. Unlike KLEE-Array, our method proposes to synergize symbolic execution and constraint solving to improve constraint solving. Our method needs to pass the information calculated information to the solver. We have compared our method with KLEE-Array on the 13 benchmark programs in Figure 7. Because KLEE-Array modifies the queries, we use the explored paths and executed instructions in symbolic execution as the metric for comparison. Table 3 shows the results.

Our method explores more paths and executes more instructions in 11 programs. On average, compared with KLEE-Array, our method achieves the relative increasing in paths and instructions as 30.31% (~12.59%~177.99%) and 40.39% (~10.33%~188.22%), respectively. These results indicate that our method is more effective than KLEE-Array on the benchmark programs. In yaml and sgml_lex, KLEE-Array is better than us because the generated constraints in these two programs are simple. KLEE-Array performs better when the constraints have few nested array expressions, and the arrays have many continuous repeated values. However, in the other 11 programs, the constraints are complex, and the array element values are diverse, on which our method outperforms KLEE-Array.

**Answer to RQ3:** Compared with KLEE-Array, our method increases the number of paths and instructions by 30.31% and 40.39%, respectively.

### 5.4 Threats to Validity

The internal threats to the validity of our work are our implementation. We alleviate the implementation problems in the design and testing phases. We carefully designed some small programs for testing and utilized KLEE’s constraint solver tool Kleaver [14] for debugging some rare constraints. Our prototype can analyze 75 real-world C programs with a wide range of scales in LoCs, which demonstrate our implementation’s robustness. The main threats are external. Although our benchmark programs are from recent symbolic execution research based on KLEE, the programs may be limited. The axiom-oriented optimizations may be specific to the

---

5We use all the KLEE’s query optimizations.
CEGAR-based solving procedure. We plan to apply our optimizations to more ABV solvers and apply the optimized solvers to more symbolic execution engines for the programs in different languages, such as SPF [32] and JDart [29] for Java programs.

6 RELATED WORK

Our work is related to program analysis and constraint solving, including constraint optimization in symbolic execution, array or bit-vector SMT theory, array or bit-vector abstraction in software or hardware verification, etc.

Improving the efficiency of constraint solving is one of the key topics in the research of symbolic execution. Many existing approaches use the SMT solver in a black-box manner and optimize the constraint before invoking the solver. KLEE [14] optimizes the constraints before solving by term rewriting, simplification, counter-example caching, and irrelevant constraint elimination. Both Green [35] and its enhanced version GreenTrie [22] propose to reuse the results of constraint solving during symbolic execution or across the symbolic execution of different programs with respect to different equivalence or implication relations. Instead of cache-based approaches, stack-based incremental solving approaches [28] are proposed to optimize the constraint solving in symbolic execution. In speculative symbolic execution [39], the symbolic executor reduces the solving invocations by speculatively executing the program under analysis. Unlike these approaches, multiplex symbolic execution [38] can utilize partial solutions to generate multiple inputs by solving once. Compared with these approaches, our approach is complementary and directly improves the underlying ABV solver’s efficiency. KLEE-Array [31] is the closest related work, which optimizes the encoding of array operations by merging the repeated values in arrays. The optimizations of KLEE-Array are on the level of symbolic execution, while our optimizations are mainly on the level of constraint solving. We have empirically compared our approach with KLEE-Array in the evaluation (c.f., Section 5.3).

Array or bit-vector SMT theory is also related to our approach. In [10], the authors investigate the complexity of the decision procedure for the combination of array theory and different theories, such as equality with uninterpreted functions (EUF) and Presburger arithmetic. CEGAR-based array constraint solving over-approximates the constraint and gradually refines the abstraction, which is adopted by modern ABV solvers [12, 19]. Besides, the idea of under-approximation is also used to find the solution faster [12] by restricting the individual bits of bit-vectors. In [5], the authors combine over-approximation and under-approximation to solve bit-vector constraints. In [36], the authors propose an interval-based method to calculate the bit information for boosting the SAT solving of bit-vector constraints. Compared with these approaches, we consider improving ABV solving under the background of symbolic execution. We integrate symbolic execution with the underlying ABV solver to improve efficiency. Besides, the work of bit-vector optimization [30] is interesting and can also be used to support a pre-checking, which is left to be the future work.

Our work is also related to the work of array or bit-vector abstraction in software or hardware verification. In [21], the authors use infeasible counter-example paths to get the predicates for array operations during the CEGAR-based verification loop. In [33], the authors propose abstraction refinement techniques to prove the quantified properties for array programs. In [15], a full-program induction technique is proposed to prove the quantified or quantifier-free properties of the programs with parametric size. In [16], the authors propose to use the program’s data and control flow information to guide the SAT solving under the background of bounded program verification. It is interesting to see whether these ideas can help ABV solving under the background of symbolic execution. For hardware verification, the work in [11, 37] uses integer linear programming to verify the hardware RTL designs, which inspires our abstraction for bit-vector constraints.

7 CONCLUSION AND FUTURE WORK

Array exists extensively in programs. The symbolic execution of array programs usually employs array SMT theory to encode the program’s array operations. Symbolic execution’s efficiency can be improved by the advancements of array constraint solving. In this paper, we propose two optimizations for CEGAR-based ABV constraint solving. The first optimization employs an ILP-based checking algorithm to check the constraint’s unsatisfiability. The other optimization removes the redundant axioms by the type information inferred during symbolic execution and the interval information computed by pre-checking. We have implemented these optimizations on the state-of-the-art symbolic executor and the ABV solver. The results of the extensive experiments on real-world benchmarks indicate that our optimizations effectively improve the efficiency of symbolic execution. The future work has the following directions: 1) apply the optimizations to other symbolic executors and solvers; 2) exploring other synergy methods between symbolic execution and constraint solving.

ACKNOWLEDGEMENTS

This research was supported by National Key R&D Program of China (No. 2017YFB1001802) and NSFC Program (No. 61632015, 62002107, 62032024, and 61690203).

REFERENCES

Type and Interval Aware Array Constraint Solving for Symbolic Execution

ISSSTA '21, July 11−17, 2021, Virtual, Denmark


