Multiplex Symbolic Execution: Exploring Multiple Paths by Solving Once

Anonymous Author(s)*

ABSTRACT

Path explosion and constraint solving are two challenges to symbolic execution's scalability. Symbolic execution explores the program's path space with a searching strategy and invokes the underlying constraint solver in a black-box manner to check the feasibility of a path. Inside the constraint solver, another searching procedure is employed to prove or disprove the feasibility. Hence, there exists the problem of double searchings in symbolic execution. In this paper, we propose to unify the double searching procedures to improve the scalability of symbolic execution. We propose *Multiplex Symbolic Execution* (MuSE) that utilizes the intermediate assignments during the constraint solving procedure to generate new program inputs. MuSE maps the constraint solving procedure to the path exploration in symbolic execution and explores multiple paths in one time of solving. We have implemented MuSE on two symbolic execution tools (based on KLEE and JPF) and three commonly used constraint solving algorithms. The results of the extensive experiments on real-world benchmarks indicate that MuSE has orders of magnitude speedup to achieve the same coverage.

KEYWORDS

symbolic execution, constraint solving, mathematical optimization

ACM Reference Format:

Anonymous Author(s). 2020. Multiplex Symbolic Execution: Exploring Multiple Paths by Solving Once. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Symbolic execution [18, 21, 38] is a precise program analysis technique that has been successfully applied to many software engineering activities, including automatic software testing [5, 41], bug finding [19], program repair [10], *etc.* One challenge of symbolic execution is the scalability problem caused by path explosion and constraint solving.

During symbolic execution, each variable in program $\mathcal{P}$ has a symbolic or concrete value. For non-branch statements, symbolic execution does symbolic or concrete calculations and updates the symbolic or concrete values of the variables. When executing a branch statement br, symbolic execution generates the path condition (PC) for each branch of br. The path condition of a branch b (denoted by $PC(b) = \bigwedge_{i=1}^{n} C_i$) is a constraint in *qualifier-free* first-order logic that encodes the feasibility of the program path to b, and C_n is the symbolic condition of b. Symbolic execution invokes a constraint solver [11, 16, 40] to check the satisfiability [23] of each branch's PC. If $PC(b)$ is satisfiable, then the program path to b is feasible, and symbolic execution will continue to execute the statement along b; otherwise, it is infeasible, *i.e.*, no input can drive the program to b, and symbolic execution abandons b. In this way, symbolic execution systematically explores the path space of $\mathcal{P}$. On one hand, the path number explodes exponentially in the number of branch statements. On the other hand, constraint solving is well-known to be hard [23]. Another complexity explosion occurs inside of the constraint solver. As shown in Figure 1, there exist double explosions in symbolic execution. Such *double explosions* obstruct the application of symbolic execution to larger real-world programs.

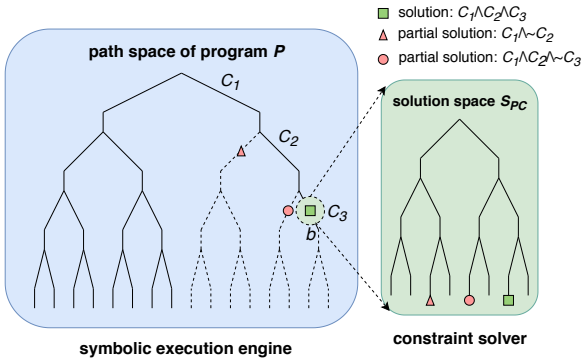


Figure 1: Double explosions in symbolic execution procedure. MuSE generates multiple test inputs from partial solutions with only one time of constraint solving. Partial solutions can be used to trigger *off-the-path* branches on the current path.

Symbolic execution explores the path space with a search strategy, such as depth-first search (DFS) and breadth-first search (BFS). The underlying constraint solver also employs an internal searching procedure in the solution space to decide the satisfiability of path conditions. Essentially, both of the path space and the solution space represent $\mathcal{P}$'s input space S_I. The conditions of $\mathcal{P}$'s branch statements split S_I into different parts. Each part can be represented by a path. For a path condition $PC(b) = \bigwedge_{i=1}^{n} C_i$, the solution space S_{PC} contains all the possible assignments to the input variables in $PC(b)$. When solving $PC(b)$, the constraint solver searches S_{PC}, and hence searches S_I. During this searching procedure, the solver may search the input space corresponding to other paths of $\mathcal{P}$. However, the solver only returns the final solution satisfying $PC(b)$, if SAT, or returns UNSAT. Therefore, S_I is doubly searched in the stack of

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

symbolic execution by the path space exploration and the underlying constraint solver. It is desirable to unify the two searching procedures to improve the scalability.

In this paper, we propose Multiplex Symbolic Execution (MuSE) towards eliminating the redundant searching in dynamic symbolic execution (DSE). The principle of our method is that we leverage the constraint solver to search the path space directly via generating multiple test inputs in one time of solving. For a path condition $PC(b) = \bigwedge_{i=1}^{n} C_i$, we call a point α in the solution space S_{PC} a *partial solution* if α satisfies a subset of the constraints in $PC(b)$. As shown in Figure 1, the solver may touch plenty of partial solutions before finding a solution or concluding the unsatisfiability. We can use partial solutions as the test inputs for exploring $\mathcal{P}$'s other paths. In this way, MuSE maps the constraint solving procedure to the path space exploration, and reduces the redundant searching to boost the whole symbolic execution procedure.

Partial solutions exist in a wide range of constraint solving algorithms. We have instantiated the idea of MuSE to three constraint solving methods to generate partial solutions: i) Simplex-based quantifier-free linear integer arithmetic (QF_LIA) constraint solving [13, 25], ii) abstraction refinement based quantifier-free array and bit-vector (QF_ABV) constraint solving [16], and iii) optimization-based floating-point constraint solving [15, 40]. Besides, we have implemented MuSE on two DSE engines based on KLEE [5] and Symbolic PathFinder (SPF) [33] for C and Java programs, respectively. We have applied our prototypes to real-world C and Java programs. The evaluation results indicate the effectiveness and efficiency of MuSE.

The main contributions of this paper are:

- We propose MuSE to utilize the partial solutions during constraint solving to generate multiple test inputs for exploring multiple paths by solving once.
- We have instantiated the idea of partial solution to three constraint solving methods and implemented MuSE on two DSE engines for C and Java programs.
- We have carried out extensive experiments on real-world C and Java programs. The experimental results indicate that MuSE achieves one or two orders of magnitude speedup on the three constraint solving methods for reaching the same code coverage.

We organize the remainders of this paper as follows. Section 2 motivates MuSE by a Simplex-based solving method. Section 3 presents MuSE and its instantiations on three solving methods. Section 4 explains the implementation of MuSE and the experiments on real-world benchmarks. Section 5 reviews the related work. Section 6 concludes the paper.

2 MOTIVATING EXAMPLE

In this section, we motivate the principle of MuSE. Figure 2 shows a Java function *start* that receives two parameters and has four paths. In each path, the program prints a different number. We call these four path as $p_1 \sim p_4$, respectively. Now we use DSE to explore the path space. Suppose that the initial input is $\{x = 1, y = 3\}$. Then the first path is p_1 that covers the lines $\{2, 3, 4, 6, 7\}$. The path condition of p_1 is $\phi_1 = x + y \geq 2 \wedge 2y - x \geq 1 \wedge 2x - y < 0$. If we use DFS searching strategy, the last branch is flipped. The new

```
1  public void start(int x,int y){ // float x, float y
2    if (x + y >= 2) {
3      if(2 * y - x >= 1) {
4        if(2 * x - y >= 0) {
5          System.out.println("#2");
6        } else {
7          System.out.println("#1");
8        }
9      } else {
10       System.out.println("#3");
11     }
12   } else {
13     System.out.println("#4");
14   }
15 }
```

Figure 2: Motivating Example

path condition $\phi_2 = x + y \geq 2 \wedge 2y - x \geq 1 \wedge 2x - y \geq 0$ is feed into off-the-shell constraint solver. Suppose that the solution of ϕ_2 is $\{x = 1, y = 1\}$, then the second path is p_2 that covers the lines $\{2, 3, 4, 5\}$. Similarly, p_3 and p_4 will be explored. In total, DSE invokes the constraint solver three times for $p_2 \sim p_4$.

Since all the constraints are linear arithmetic, we suppose that the solver uses the Simplex-based QF_LIA theory solving algorithm [23, 25]. The algorithm first considers all the integer variables in the constraints as real variables and uses Simplex-based linear real arithmetic solving algorithm to solve the constraints. If there is no solution, the constraints are unsatisfiable. If there exists a solution and the values in the solution are already integers, the algorithm returns the solution; otherwise, the algorithm adds the integer requirement constraints gradually and employs Simplex procedures again to find the integer solution.

The Simplex algorithm maintains an assignment α to store the values of variables[1]. If the assignment does not satisfy the constraints, the algorithm changes the assignment so that at least one unsatisfied constraint becomes true. This procedure continues until all the constraints are satisfied or returns UNSAT. For example, suppose that the path condition is $\phi_2 = x+y \geq 2 \wedge 2y-x \geq 1 \wedge 2x-y \geq 0$ and the initial assignment is $\alpha_0 = \{x = 0, y = 0\}$. Since α_0 does not satisfy $x + y \geq 2$ and $2y - x \geq 1$, Simplex changes the assignment to $\alpha_1 = \{x = 2, y = 0\}$ by the so-called *pivot* operation [23, 25], such that $x + y \geq 2$ is satisfied. Now Simplex validates assignment α_1 and finds that $2y - x \geq 1$ is violated. In the next step, the pivot operation changes assignment α_1 to $\alpha_2 = \{x = 1, y = 1\}$. Finally, all the constraints are satisfied and α_2 is already an integer solution. So, α_2 is returned to the DSE engine for generating the input for path p_2. In vanilla DSE, the constraint solver is used as a black box, only α_2 is visible to the DSE engine, and the DSE engine generates only one test case from one time of constraint solving.

In contrast, MuSE uses the constraint solver in a white-box manner. As shown in Figure 3, the input space of *start* (x-y plane) is split into 7 parts by the three lines corresponding to the three branch statements. Each of paths $p_1 \sim p_3$ corresponds to one part

[1]Details of Simplex algorithm is discussed in Section 3.2.

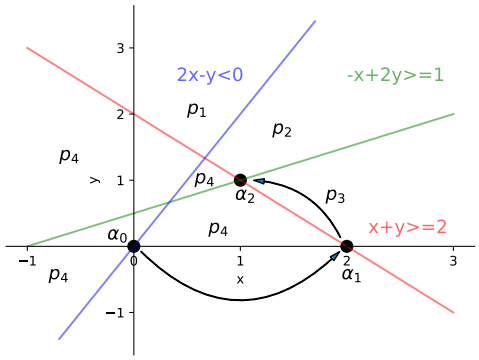


Figure 3: Branch statements split the input space into different parts corresponding to different paths. The black arrows show how Simplex searches the solution space for $\phi_2 = x + y \geq 2 \wedge 2y - x \geq 1 \wedge 2x - y \geq 0$. The solving procedure covers three points corresponding to the paths p_4, p_3 and p_2, respectively.

and p_4 corresponds to four parts. Simplex algorithm leverages the linear property of the constraints and smartly explores the solution space. We can say that Simplex algorithm is exploring the path space of the program. Since the intermediate assignments α_0 and α_1 satisfy subsets of the constraints in ϕ_2, they can trigger p_4 and p_3, respectively. These intermediate assignments are *partial solutions*. DSE can utilize these partial solutions to steer the exploration along off-the-path branches on the current path. For example, when solving the first path condition ϕ_2, MuSE generates two extra inputs from the partial solutions α_0 and α_1, and the executions of these two inputs trigger p_4 and p_3, respectively. Hence, by utilizing partial solutions, MuSE *only needs one time of constraint solving to explore all the paths*.

With the support of partial solutions, MuSE maps the constraint solving procedure to the path exploration in DSE by releasing the power of constraint solver.[2] The key requirement of MuSE is that the underlying constraint solver can generate partial solutions. Actually, partial solutions widely exist in the current constraint solving methods (*c.f.* Section 3.5). Besides, we will see in the experiments (*c.f.* Section 4) that MuSE can generate hundreds of partial solutions with one time of constraint solving in practice.

3 METHOD

In this section, we first show how MuSE works with dynamic symbolic execution framework. Then we elaborate on how to generate partial solutions in the existing constraint solving algorithms.

3.1 DSE With MuSE

Algorithm 1 shows the procedure of DSE with MuSE. The inputs are the program P and an initial input seed I_0. T stores all the generated

[2]Multiplex means reusing a shared scarce resource by sending multiple messages at once, which is analog to exploring multiple paths by solving once. So we call our method Multiplex Symbolic Execution (MuSE).

Algorithm 1 Multiplex Dynamic Symbolic Execution

```
1:  Input: Program P, initial input seed I_0
2:  T = {I_0} //test cases to be executed
3:  B = ∅ // open branches to be explored
4:  while T ≠ ∅ do
5:      I = select(T)
6:      p = concolicExecute(P, I)
7:      saveUnexploredBranches(p, B)
8:      b = searchStrategy(B)
9:      φ = pathCondition(b)
10:     (res, solution, partial-solutions) = solving(φ)
11:     if res = sat then
12:         T = T ∪ {solution}
13:     end if
14:     /*save partial-solutions whether SAT or not, if any*/
15:     T = T ∪ partial-solutions
16:     if stopCriterion() then
17:         break
18:     end if
19: end while
```

test inputs yet to be executed. The *while* loop selects a test input I from T and executes the program in a concolic manner [18, 38] (line 6). The function *saveUnexploredBranches* saves all the unexplored branches on the current path p into B (line 7). Then one branch b is selected from B according to a search strategy (line 8). Here any strategies can be used to prioritize the branches in B, such as DFS and BFS. Then the function *pathCondition* generates the path condition ϕ along b (line 9). The key of our method is that the algorithm uses an extended constraint solver which returns a triple (*res*, *solution*, *partial-solutions*) (line 10). When *res* is SAT, *solution* is the target test input that can steer the execution along b. Then the *solution* is stored into T for future executions (line 12). Otherwise, *res* is UNSAT or UNKNOWN, *solution* is set as *null*. We assume that the underlying constraint solver may generate partial solutions no matter whether the final solution can be found or not. Therefore, *partial solutions* are also stored into T, if any (line 15). So, the DSE procedure can get multiple inputs by invoking the constraint solver once. Even for an unsatisfiable path condition, the expensive computation spent on constraint solving would not be wasted.

The key to the success of our algorithm is that the constraint solver can generate partial solutions. A wide range of constraint solving algorithms conform a *trial and error* mode. Thus we can extract plenty of partial solutions from the solving procedure. In the following of this section, we firstly focus on several commonly used constraint solving algorithms in symbolic execution. Then we briefly discuss more constraint solving algorithms. We will see that the notion of partial solution is a universal principle applicable to a wide range of constraint solving algorithms.

3.2 Partial Solutions in Simplex

Simplex is an old and efficient constraint solving method for linear arithmetic [13, 25]. Although the worst case of Simplex's complexity

Algorithm 2 General Simplex with Partial Solutions

```
1: Input: A set of constraints in linear real arithmetic
2: Transform S into general form
3: initialize assignment α
4: partial-solutions = ∅
5: while True do
6:   if no basic variable violates bounds then
7:     return (SAT, α, partial-solutions)
8:   else
9:     partial-solutions = partial-solutions ∪ {α}
10:    if can find x_i and x_j for pivoting then
11:      pivot(x_i, x_j)
12:      update α
13:    else
14:      return (UNSAT, null, partial-solutions)
15:    end if
16:  end if
17: end while
```

is exponential, it is widely used in practice. For example, one of the state-of-the-art SMT solver Z3 [11] uses Simplex.

Algorithm 2 shows how partial solutions can be supported by the basic procedure of general Simplex algorithm for quantifier free linear real arithmatic (QF_LRA). The inputs of the algorithm is a set of m linear constraints S. Without considering the preprocess of the constraint solver, we assume that S corresponds to m path constraints in ϕ. At the beginning of the procedures, the i-th constraint $\Sigma_{x_j \in N} a_{ij} x_j \bowtie R_i$ ($\bowtie \in \{=, \geq, \leq\}$) [3] is transformed into the general form $\Sigma_{x_j \in N} a_{ij} x_j - x_i = 0 \wedge x_i \bowtie R$, where x_i and x_j are real variables. For example, $x + y \geq 2$ is transformed into $x + y - s = 0 \wedge s \geq 2$. After the preprocessing, the constraint system S is in the general form

$$Ax = 0 \textbf{ and } \bigwedge_{i=1}^{m} l_i \leq x_i \leq u_i \tag{1}$$

where A is the $m \times (n + m)$ coefficient matrix and $x_i \in B$. The elements of B and N are called basic variables and non-basic variables with the real number set as domains. The algorithm represents A with a tableau where the columns and rows correspond to non-basic and basic variables respectively.

Simplex algorithm maintains an assignment $\alpha : B \cup N \rightarrow Q$ where Q is rational numbers set. Initially, all variables are set to zero (line 3). At line 6, the algorithm checks whether all the bounds in equation 1 are satisfied. If yes, the algorithm returns the current assignment as the solution. Otherwise, the current assignment α violates at least one bound in equation 1. Here we also know which constraints in ϕ are satisfied. Suppose that the first k constraints are satisfied, and the $k+1$ constraint is not. The current assignment α can be used as the test input steering the execution along the $(k+1)$-th open branch in the current path p. Therefore, the current assignment is stored in the partial solution set (line 9). Simplex is both sound and complete. In the next step, the algorithms checks whether there exist a basic variable x_i and a non-basic variable x_j can be pivoted. If not, the constraint system S is unsatisfiable, and the algorithm returns UNSAT and the current partial-solutions. Otherwise, the Simplex algorithm changes A in the *pivot* operation. Here x_j is solved in the row i as

$$x_j = \frac{x_i}{a_{ij}} - \sum_{x_k \in N - \{x_j\}} \frac{a_{ik} x_k}{a_{ij}}, \ (a_{ij} \neq 0) \tag{2}$$

In all other rows except row i, x_j is replaced by equation 2 such that x_j becomes basic variables and x_i becomes non-basic variables. Here the assignment $\alpha(x_j)$ is changed so that x_i satisfies its bounds.

As the *while* loop continues, more and more bounds in equation 1 may be satisfied. Therefore, we can record the current assignment before each pivot operation as a partial solution.

In this paper, we consider QF_LIA constraints, which can be solved by a Simplex-based method [23]. The procedure first considers the variables as real variables and use Simplex to solve the constraint. If no solution, the constraint is unsatisfiable. If there is a solution, the procedure return the solution if all the variables have integer values. Otherwise, the procedure selects a variable that has a non-integer value and add the constraints of integer requirements. Then, the procedure employs Simplex again to search the integer solution. Hence, Simplex is the underlying searching procedure for solving QF_LIA. Solving a QF_LIA constraint may invoke Simplex procedure multiple times. Hence, we record partial solutions for QF_LIA constraint solving in the underlying Simplex procedures.

3.3 Partial Solutions in Array Theory Solving

Arrays widely exist in programs. Existing symbolic executors [5, 6] use array theory for representing the array operations in programs. Reasoning about array can be very complex when both index and array elements are symbolic values. In this subsection, we focus on how to generate partial solutions in array theory solving.

Abstract/refinement-based array theory solving [16] is the state-of-the-art solving method. Several mainstream QF_ABV SMT tools implements this method, such as STP [16] and Boolector [31]. It is natural to generate partial solutions in the abstraction/refinement loop. Algorithm 3 shows the procedure. At the beginning, formula f is converted to an abstract version f_a by the function *abstract* (line 3). In fact, f_a is a relaxation of f, where the constraints derived from array axioms [23] are omitted, so f implies f_a. Here we assume that the constraints are expressed in bit-vector theory. The algorithm feeds f_a into a SAT-based bit-vector solver. If f_a is unsatisfiable, then f is unsatisfiable too. Otherwise, the algorithm validates the solution M_a of f_a on f. If M_a is also a solution of f, then the algorithm returns M_a as the solution. Otherwise, f_a is refined so that more constraints derived from array axioms are added as conjunctives to f_a (line 18). The loop continues until the algorithm terminates. The abstract version f_a becomes f if all constraints derived from array axioms are added into f_a. So the algorithm always terminates.

It is straightforward to extract partial solutions from this abstract/refinement loop. In each loop iteration, we store the solution M_a of the abstract formula f_a as partial solution (line 13). In fact, we can also modify the SAT-based bit-vector solver to obtain more partial solutions at line 5. We will discuss this topic in subsection 3.5.

[3] Strict inequalities and disequalities are processed by additional tricks. More details can be referred to [13].

Algorithm 3 Abstract/Refinement-based Array Theory Solving with Partial Soltions

```
1: Input: A conjunction of formulas f in bit vector and array theory
2: partial-solutions = ∅
3: f_a = abstract(f)
4: while true do
5:    (res, M_a) = BVSolver(f_a)
6:    if res = UNSAT then
7:       return (UNSAT, null, partial-solutions)
8:    else
9:       v = validate(M_a, f)
10:      if v=true then
11:         return (SAT, M_a, partial-solutions)
12:      else
13:         partial-solutions = partial-solutions ∪ {M_a}
14:         f'_a = refine(f_a)
15:         if f'_a = f_a then
16:            return (UNSAT, null, partial-solutions)
17:         else
18:            f_a = f'_a
19:         end if
20:      end if
21:   end if
22: end while
```

3.4 Partial Solutions in Optimization-based Solving

Floating-point arithmetic is a challenge for constraint solving. Recently, mathematical optimization based solving has been proposed for floating-point arithmetic, such as XSAT [15] and CORAL [40]. Optimization-based constraint solving introduces the techniques from search-based testing, where the constraints are transformed into an objective function for the optimization algorithms. The global minimum of the objective function corresponds to the solution to the solving problem. Thus, any mathematical optimization algorithms can be used. For example, XSat uses Monte Carlo Markov Chain (MCMC) method to find the global minimum [15]. CORAL uses meta-heuristic algorithms, including random search or particle swarm optimization [40]. In this subsection, we take XSat as an example to show how partial solutions can be supported in optimization-based constraint solving algorithms.

XSat transforms the constraints solving problem as follows. Given a conjuctive normal form of constraint:

$$F = \bigwedge_{j \in J} \bigvee_{i \in I} e_{i,j} \bowtie e'_{i,j} \tag{3}$$

the objective function (or fitness function) corresponding to F is

$$O_F = \sum_{j \in J} \prod_{i \in I} d(\bowtie, e_{i,j}, e'_{i,j}) \tag{4}$$

where d is defined in table 1.

Here $\theta(x, y)$ is the number of floating-point numbers between a and b, representing the distance between x and y. The fitness function evaluates how close a test input satisfies the constraints. Since $\theta \geq 0$, O_F equals to 0 if and only if F is satisfied. For example,

Table 1: Fitness function

$d(==, x, y)$	$\theta(x, y)$
$d(\leq, x, y)$	$x \leq y \; ? \; 0 : \theta(x, y)$
$d(\geq, x, y)$	$x \geq y \; ? \; 0 : \theta(x, y)$
$d(<, x, y)$	$x < y \; ? \; 0 : \theta(x, y)$
$d(>, x, y)$	$x > y \; ? \; 0 : \theta(x, y)$
$d(\neq, x, y)$	$x \neq y \; ? \; 0 : 1$

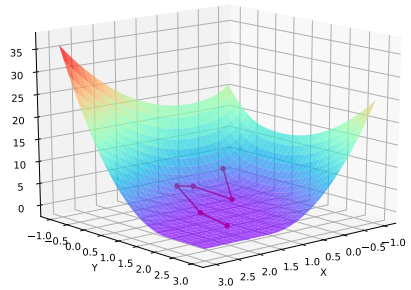


Figure 4: Landscape of the fitness function corresponding to path condition $\phi_2 = x + y \geq 2 \wedge 2y - x \geq 1 \wedge 2x - y \geq 0$. MCMC finds the global minimum after 5 steps.

suppose that x and y in Figure 2 are floating-point variables. The landscape of the objective function for path condition $\phi_2 = x + y \geq 2 \wedge 2y - x \geq 1 \wedge 2x - y \geq 0$ is shown in Figure 4. Note that the landscape of more complex path constraints can be very non-convex. There is no guarantee that the global minimum can always be found.

Given the objective function, we can use any mathematical optimization methods to find the global minimum, including simulated annealing, particle swarm Optimization, *etc.* For example, XSat uses simulated annealing algorithm to find the global minimum. Simulated annealing specializes the general MCMC sampling method by using symmetric proposal distribution and $exp(-f(x)/T)$ as the density of target unnormalized distribution, where f is the objective function and T is the temperature [22]. In principle, we can extract the local minimum or the intermediate points as partial solutions directly. As shown in Figure 4, we get the final solution $(1.68, 2.06)$ in 5 steps, and also two partial solutions $(0, 0)$, $(1.45, 1.06)$ corresponding to path $p4$ and $p3$ respectively. Besides, the mathematical optimization procedure may produce many partial solutions, and some of them may trigger the same path. We can choose to extract partial solutions only from local minimums, which are supported by modern optimization algorithms. For example, the Basin-Hopping algorithm extends MCMC method by employing a local optimization method in each Monte Carlo step [32]. Note that, due to the stochastic nature of MCMC methods, it may take different steps before the final solution is found in multiple runs. Hence, we may get different partial solutions in different runs for the same constraint.

3.5 The Ubiquitous Partial Solutions

Until now, we have discussed how partial solutions can be supported in three commonly used constraint solving algorithms in

symbolic execution. In practice, symbolic execution barely generates disjunction in path condition, which is a hurdle to the general DPLL(T) framework in SMT solvers [23]. Therefore, in this paper, we briefly discuss how partial solutions can be supported in several other constraint solving algorithms. Here we take some of them as instances.

CDCL framework for SAT. The popular CDCL framework used in SAT solvers searches the solution space by a decide-backtrack loop [23]. CDCL maintains a partial assignment to variables. In each loop iteration, the decide phase chooses an assignment to a selected variable, then the backtracking phase erases some assignments to resolve conflicts. The loop continues until the partial assignment is extended to a full assignment or return UNSAT. Obviously, we can extract the partial assignments as partial solutions. Besides, fixed-sized bit-vector theory solving uses SAT solving as the underlying solver. Nowadays, bit-vector theory [23] is widely used in symbolic execution to precisely model the computations of machine numbers in programs. Many mainstream SMT solvers support bit-vector theory, such as Z3 [11] and STP [16]. Bit-vector theory solving converts a bit-vector constraint to a propositional formula and invokes the underlying SAT solver for solving. Therefore, we can record the partial solutions during the SAT solving procedure and generate partial solutions for bit-vector solving. Hence, the method in Section 3.3 for QF_ABV can be improved further, which is left to be the future work.

DPLL(T) for Satisfiability Modulo Theories (SMT). DPLL(T) framework extends the CDCL framework with the decision procedures of background theories [23]. DPLL(T) uses CDCL algorithm to find a solution to the propositional skeleton of the constraints and then employs a constraint solver of background theories (*e.g.*, Simplex for linear arithmetic) to find a solution. We can extract partial solutions as in CDCL framework and constraint solving algorithms for background theories.

Congruence-Closure algorithm for equality logic and uninterpreted functions. The Congruence-Closure algorithm constructs the congruence-closed equivalence classes for terms [23]. If a disequality constraint violates the equivalence relation, the algorithms return UNSAT. We can construct partial solutions according to the equivalence classes at any point during the construction procedure.

JFS is a recent constraint solving algorithm for floating-point arithmetic [28]. JFS transforms floating-point constraints into a program having a list of branch statements. The inputs satisfy the constraints only when the final statement of the program is covered. Thus, the constraint solving problem is transformed into a statement covering problem. JFS employs coverage-guided fuzzing [1] to generate the test cases to cover the target statement. Therefore, JFS is similar to the optimization-based floating-point solving methods where the optimization is implemented implicitly by the fuzzing method. We can extract partial solutions as like in Section 3.4.

3.6 Discussion

MuSE is the first step towards unifying the double searching procedures in symbolic execution. Compared to vanilla symbolic execution, MuSE can generate more test inputs with the same amount of constraint solvings, *i.e.*, exploring multiple paths by solving once. We have no guarantee that each of the generated test inputs by partial solutions can trigger a distinct path. From the results in section 4, we will see that multiple partial solutions indeed correspond to the same path. A potential method to avoid expensive symbolic execution with equivalent partial solutions as inputs is to employ lightweight concrete executions on these partial solutions. Such advanced synergy of concrete and symbolic execution can both exploit partial solutions and avoid expensive redundant symbolic computations. We leave this as the future work.

4 EXPERIMENTAL EVALUATION

4.1 Implementation

In order to support partial solutions, we have extended the three constraint solving methods discussed in section 3 on two state-of-the-art constraint solvers and one self-implemented solver.

- We use Z3 [11] for QF_LIA solving and modified Z3 to generate partial solutions. Z3 employs Simplex-based QF_LRA solving to find the real solution to a relaxed problem first and then adds constraints gradually to find integer solutions [12, 26]. We record the partial solutions in the iterations of Simplex-based solving and convert each partial solution to its closest integer version.
- We use STP [16] for QF_ABV solving. STP implements the abstract refinement-based approach for solving QF_ABV formulas. We record the partial solution generated in STP's each refinement iteration.
- We have implemented an optimization-based floating-point solver in Java. We use simulated annealing [22] as the optimization algorithm. In each Monte Carlo step, we check the assignments and extract them as partial solutions if needed.

To evaluate MuSE extensively, we built the DSE engines utilizing partial solutions for C and Java programs based on the state-of-the-art symbolic executors.

- We have implemented MuSE on our DSE engine [34, 44] for C programs based on KLEE [5]. The engine uses the STP enabled for providing partial solutions. Besides, the DSE engine supports the under-constrained symbolic execution [35] that can easily carry out the DSE for a function.
- We have implemented MuSE on our DSE engine [46] for Java programs based on SPF [33]. The engine uses the Z3 supporting partial solutions for solving QF_LIA formulas and our optimization-based solver for analyzing floating-point programs.

4.2 Research Questions

We carry out experiments to answer the following questions:

- Effectiveness: How effective is MuSE to test a program automatically compared with the existing search strategies?
- Efficiency: How efficient is MuSE to accomplish a testing task compared with the existing search strategies?

4.3 Setup

We conduct three experiments to evaluate MuSE with the three constraint solving methods. The setups of these experiments are as follows.

Table 2: The programs in the QF_LIA experiment.

Programs	LOC	Brief Description
BMPDecorder	266	BMP file decoder
AviParser	2046	ImageJA AVI file decoder
GifParser	439	ImageJA GIF parser
BMPParser	205	ImageJA BMP parser
PGMParser	2736	ImageJA PGM parser
ImgParserPCX	945	Imaging PCX decoder
ImgParserBMP	1123	Imaging BMP decoder
JaadParser	115	Jaad MP3 decoder
Schroeder	1448	Schroeder WAV decoder
JMP3Parser	1634	JavaMP3 decoder
Toba	1060	Java bytecode decoder
Total	**12017**	**11 open source programs**

4.3.1 Experiment 1: Simplex-based QF_LIA Solving. We use the DSE engine for Java programs and Z3 enabled with partial solutions during QF_LIA solving for evaluation. Table 2 shows the benchmark programs. All the programs are real-world open-source Java programs. The programs are file parsing libraries of different kinds of file formats, including BMP, MP3, WAV, *etc.* Four programs are from ImageJA library, which is a popular library for manipulating images, and the total LOC of the library is 123783. Two programs are from apache imaging library, whose LOC is 32594. For each program, we count the LOC of the files that are directly related to the tested parsing interface.

We create a driver program for each parsing library program to invoke its main parsing interface. We use a valid file as the initial input. We symbolize each byte in the file for DSE. Because many bit operations exist in these programs, and QF_LIA does not support the modeling of bit operations, we convert bit operations to its integer-implemented version. We use DSE to test these programs automatically in four configurations: DFS, DFS with partial solutions (DFS+P), BFS, and BFS with partial solutions (BFS+P). Since the initial inputs are valid files and cover a large portion of instructions, we use the number of the new instructions covered after the first path as the criterion to evaluate methods. We test each program in each configuration for 15 minutes.

4.3.2 Experiment 2: Abstraction Refinement-based Array Theory Solving. We use the DSE engine for C programs and the partial solution enabled STP for evaluation. Table 3 shows the benchmark programs, which are from the GNU scientific library (GSL) [14]. Floating-point operations and array operations are extensive in these programs. The second column **LOC** displays the total number of the LLVM instructions inside the tested function and its callee functions.

To support floating-point operations, we have implemented the method in [36] that converts each floating-point instruction to its integer simulation implementation. We use softfloat [3] as the library for floating-point simulation. We test these programs in six configurations: DFS with partial solutions (DFS+P), BFS with partial solutions (BFS+P), the default random-cover new search (RCN), the random state search (RSS), DFS, and BFS. We test each program in each configuration for 15 minutes.

Table 3: The programs in the QF_ABV experiment. LOC*: lines of LLVM codes.

Programs	LOC*	GSL Function
akimaei	2225	akima_eval_integ
bilinea	2075	bilinear_deriv_y
find	7706	find
eigengs	9150	gsl_eigen_genv_sort
fft-rrt	10199	gsl_fft_real_radix2_transform
h2d-ps	7919	gsl_histogram2d_pdf_sample
sort	328	gsl_sort
sum-lum	2896	gsl_sum_levin_u_minmax
linear-ed	1618	linear_eval_deriv
linear-ei	2117	linear_eval_integ
solve-ct	8066	solve_cyc_tridiag
solve-ctn	8205	solve_cyc_tridiag_nonsym
steffen-ei	1815	steffen_eval_integ
Total	**64319**	**13 GSL functions**

Table 4: The programs in the experiment for optimization-based floating-point solving.

Programs	LOC	Brief Description
EigenD	1985	La4j Eigen decomposition
JacobiS	1210	La4j Jacobi solver
CholeskyD	1203	La4j Cholesky decomposition
LeastS	1264	La4j least squares solver
SquareR	1224	La4j square root solver
EDAnalysis	1429	Colt Eigen value decomposition
Mutil	1120	Colt linear matrix multiplication
RankAnalysis	1120	Colt rank for matrix
SVDAnalysis	1139	Colt singular value decomposition
TVSAnalysis	1120	Colt several kinds of decomposition
Total	**12814**	**10 open source programs**

4.3.3 Experiment 3: Optimization-based Floating-point Solving. Similar to experiment 1, we also evaluate MuSE equipped with the partial solution enabled optimization-based floating-point constraint solver on the DSE engine for Java programs. Table 4 shows the benchmark Java programs. All the programs are from two real-world linear algebra Java libraries (*i.e.*, La4j and Colt), which include very complex floating-point arithmetics. Same as Table 2, we count the LOC of each benchmark's directly related files. The LOCs of Colt and La4j are 32879 and 10963, respectively.

We reuse the testing drivers that already exist in the libraries. The drivers usually input double typed matrices to the library's interfaces. We symbolize each element in the matrixes to test these programs.

All experiments are carried out on a server with 64GB RAM and one 3.4GHz Xeon CPU with six cores. The results are the average of 3 runs.

4.4 Experimental Results

This sub-section illustrates the results of the three experiments evaluating MuSE.

4.4.1 Experiment 1: Simplex-based LIA Solving. Table 5 gives the experimental results of evaluating MuSE on QF_LIA constraint solving. The column **#T** displays the number of the tests generated by DSE. The column **#NI** displays the number of new instructions that have been covered after the first path. Both these two numbers reflect the DSE's ability of path exploration. As shown by the table, compared with the baseline search strategy, *i.e.*, DFS or BFS, MuSE can cover more new instructions and generate more tests for most of the programs, which indicate the effectiveness of MuSE.

Figure 5 shows the trends of covered new instructions under different configurations for all the programs. The X-axis is the analysis time, and the Y-axis shows the number of newly covered instructions after the first path. As shown in the figure, under the same period, MuSE covers more new instructions consistently than the baseline search strategy. We also evaluate the efficiency of MuSE with the time needed to cover the same amount of new instructions. DFS achieves its largest number of new instructions (*i.e.*, 398) at 625.6s, and DFS+MuSE covers the closest number (*i.e.*, 400) at 14.4s and achieves at least 43.4x speedup. On the other hand, MuSE achieves at least 12.9x (890.6s/69.2s) speedup on BFS for covering 1296 instructions that are no less than BFS's upper bound (*i.e.*, 1296). These results indicate that MuSE is highly efficient.

4.4.2 Experiment 2: Abstraction Refinement-based Array Theory Solving. Table 6 gives the experimental results. Column **#PS** shows the number of partial solutions generated during analysis. Column **COV** shows the coverage result. As shown by the table, compared with the other configurations, **BFS+P**, *i.e.*, MuSE plus BFS, achieves the best result in average. Compared with the last four strategies in KLEE, **BFS+P** improves more than 20% coverage, which indicates the effectiveness of MuSE.

Figure 6 shows the coverage trend under different strategies. The X-axis shows the analysis time, and the Y-axis shows the achieved average coverage. As shown in the figure, MuSE performs consistently better than the baseline strategies. The results also indicate that MuSE has two orders of magnitude speedup to achieve the upper bound coverages of DFS and BFS, respectively.

4.4.3 Experiment 3: Optimization-based Floating-point Solving. Table 7 shows the experimental results of MuSE on optimization-based floating-point solving. The meanings of the columns are the

Table 5: Experimental Results of QF_LIA

Programs	DFS+P		DFS		BFS+P		BFS	
	#T	#NI	#T	#NI	#T	#NI	#T	#NI
BMPDecorder	1125	134	5	0	3746	84	104	40
AviParser	340	117	144	46	1732	101	114	0
GifParser	721	25	60	5	1905	64	960	48
BMPParser	1203	52	8	0	4458	126	102	18
PGMParser	264	1	263	1	4736	188	7362	178
ImgParserPCX	387	38	81	20	2596	76	65	0
ImgParserBMP	458	314	114	21	1784	528	135	198
JaadParser	2083	64	134	0	2692	64	2835	59
Schroeder	1149	23	235	20	2267	29	402	22
JMP3Parser	214	286	37	198	319	653	279	646
Toba	1836	344	117	87	1670	311	179	87
Average	889	127	108	36	2536	202	1139	117

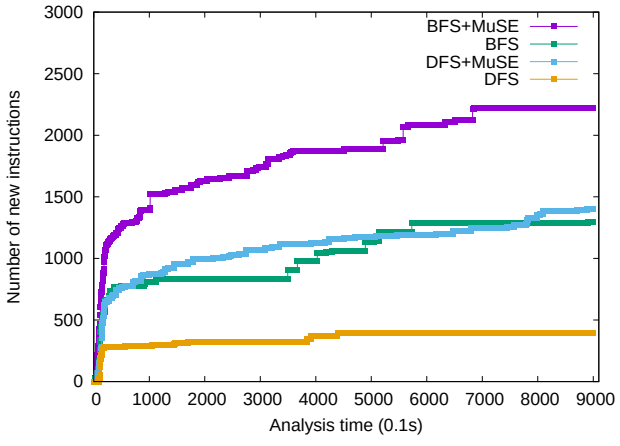


Figure 5: Trends of number of covered new instructions in experiment on QF_LIA.

Table 6: Experimental Results of QF_ABV

Programs	DFS+P		BFS+P		Other Stategies			
	#PS	COV	#PS	COV	RCN	RSS	DFS	BFS
akimaei	1	64.7	514	76.1	76.5	67.2	65.3	64.9
bilinea	305	71.6	172	80.8	79.0	77.4	59.1	65.4
find	177	96.9	156	96.7	91.3	40.0	91.5	97.7
eigengs	19	73.5	118	98.0	67.6	51.6	61.1	82.8
fft-rrt	1015	46.8	350	99.5	39.6	38.6	46.5	11.3
h2d-ps	4	95.7	130	98.6	47.5	47.5	95.7	98.6
sort	18	100.0	9	100.0	89.7	82.2	83.7	44.6
sum-lu	29	76.5	129	88.6	70.8	50.7	70.1	43.1
linear-ed	13	63.1	1015	82.8	79.9	78.7	56.3	63.8
linear-ei	3	73.5	376	80.5	77.5	71.2	64.2	72.6
solve-ct	135	93.4	33	94.4	26.6	22.3	13.8	93.5
solve-ctn	32	94.2	2	96.0	21.7	19.2	30.0	95.5
steffen-ei	18	74.8	253	83.4	68.7	65.6	67.4	68.8
Average	136	78.8	250	90.4	64.3	54.8	61.9	69.4

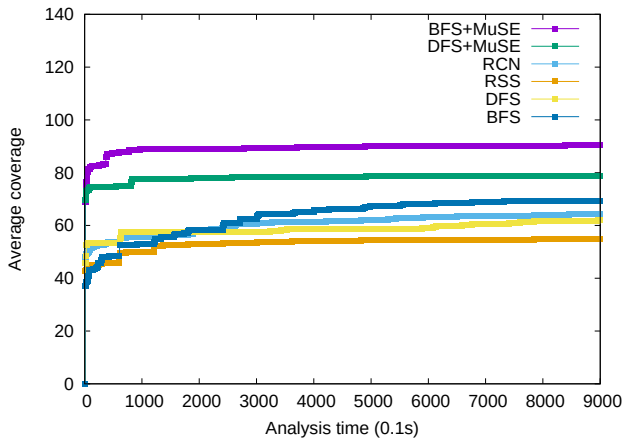


Figure 6: Trends of coverage in experiment on QF_ABV.

same as those in Table 5. Compared with the baseline method, MuSE covers more instructions and generates more tests. Besides, compared with BFS, MuSE improves DFS more. The reason is that DFS explored longer paths in priority. Longer paths have more complex PCs that are hard to resolve. On the other hand, BFS explores short paths first, whose PCs are more likely to be solved by optimization-based solvers. The results marked with an asterisk in Table 7 mean that the analyses terminate before the time limit. This is because the path conditions are too complex for the constraint solver. Symbolic execution can not expand the execution tree when the constraint solver can not generate solutions.

Figure 7 shows the trend of newly covered instructions under different configurations. The X-axis is the analysis time, and the Y-axis shows the number of newly covered instructions for all the programs after the first path. As shown by the figure, MuSE is consistently more efficient than the base line method. Besides, MuSE with DFS finally covers more instructions than BFS, which also indicates that partial solutions improve the effectiveness. Similar

Table 7: Experimental results of optimization-based floating-point solving. Results marked with asterisk mean that symbolic execution can not generate more inputs for complex path conditions and stops expanding the execution tree before time limit at that run.

Programs	DFS+P		DFS		BFS+P		BFS	
	#P	#NI	#P	#NI	#P	#NI	#P	#NI
EigenD	3	244	1	0	477	1028	20	965
JacobiS	1424	13	43	6	1151	13	43	6
CholeskyD	1376	1335	43	4	1116	8	42	8
LeastS	169	2000	1	0	573	2246	43	2196
SquareR	1541	166	43	4	1240	8	44	8
EDAnalysis	8	418*	3	3*	8	392*	3	3*
Mutil	10	7*	4	0*	10	7*	4	0*
RankAnalysis	255	406	15	180	325	427*	20	427*
SVDAnalysis	204	427*	38	418*	276	427*	19	427*
TVSAnalysis	343	430	1	0	484	612	7	225
Average	484	495	17	55	514	469	22	387

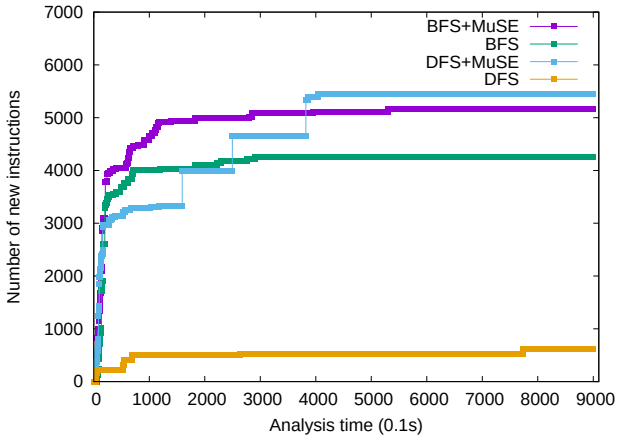
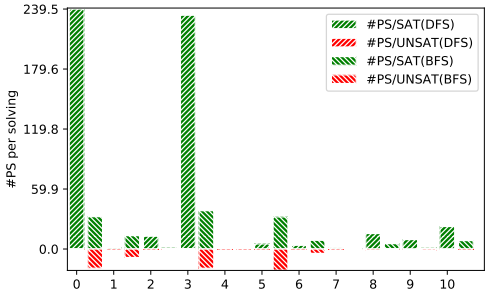
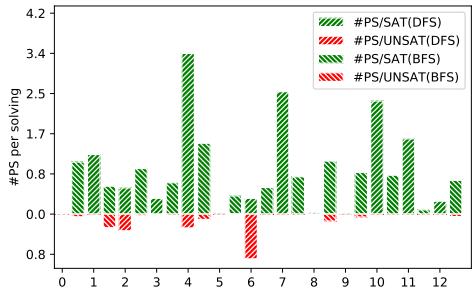


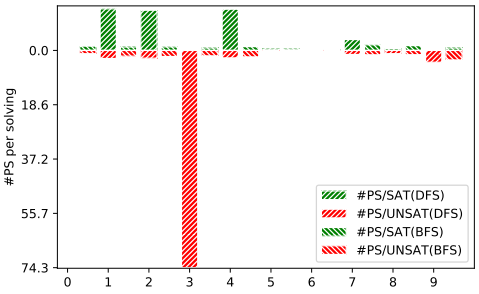
Figure 7: Trends of number of covered new instructions in experiment on floating-point solving.



(a) programs for QF_LIA



(b) programs for QF_ABV



(c) programs for floating-point

Figure 8: The average number of partial solutions (#PS) per solving. The x-axes are the programs used in experiments.

to experiment 1, we also evaluate the efficiency of MuSE by comparing it with the baseline for covering the same amount of new instructions. To achieve DFS's largest number of new instructions (*i.e.*, 615 at 773.1s), MuSE uses 7.8s to cover 633 instructions and gets 99.1x speedup. For BFS, MuSE gets at least 4.6x (290.2s/63.2s) speedup for covering 4350 instructions that are closest to BFS's upper bound (*i.e.*, 4265). These results indicate the efficiency of MuSE.

4.4.4 Partial Solutions. We also collect the results of constraint solving and the numbers of partial solutions. Figure 8 shows the average number of partial solutions per solving in different experiments. Due to the difference between constraint solving methods,

the number of partial solutions can be very different. For example, in one analysis on BMPDecorder using QF_LIA under DFS mode, our method generates 923 partial solutions in 4 times of solvings. For fft-frrt using QF_ABV under DFS mode, our method generates 940 partial solutions in 232 times of solvings. It worth noting that our method usually generates more partial solutions under DFS than BFS. The reason is that the path condition under DFS contains more constraints and is usually more complicated, so the solving algorithms need more trials before finding the final solution.

4.5 Threats to Validity

The threats to the validity of the experimental results are mainly external. Although we only applied the idea of partial solutions on three solving methods, the idea is general and can be applied to other constraint solving methods (*c.f.*, Section 3.5). The benchmarks we use for evaluating MuSE on three constraint solving methods are limited, and thus the experimental results could be biased. We plan to evaluate MuSE on other constraint solving methods and benchmark programs for a more general validation. The internal threats mainly come from the bugs in implementation due to the complexity of the solvers and the DSE engines. We have designed a set of test cases to test the partial solution enabled solvers and the DSE engines utilizing partial solutions.

5 RELATED WORK

The key idea of MuSE is that the underlying constraint solver supports partial solutions. We expect that the mainstream constraint solvers to provide a general interface to access partial solutions in the future.

The most related work to our method is the constraint optimization techniques. In many symbolic execution engines, solving result cache stores the previous solving results so that the same constraints are not repeatedly solved [5, 6]. The counter-example cache also stores the sets of unsatisfiable constraints. When the query contains a subset stored in the counter-example cache, there is no need to invoke the constraint solver [5]. Both of symbolic execution and constraint solving reduce constraints into simpler forms before querying the underlying constraint solver. For example, KLEE rewrites expressions by folding constants and simplifying linear expressions [5]. EXE separates a query into independent subsets of constraints so that the solving result cache can be reused better. The Green framework provides a unified facility so that the constraint solution can be reused across multiple programs and analysis [42]. Green also canonicalizes constraints to improve the cache hit ratio. In [20], Green is extended to support logical implication relations between constraints. Speculative symbolic execution (SSE) executes branch statements speculatively [48]. The constraint solver is invoked until a specified number of constraints is collected on the current path. The total invocation times of constraint solver is reduced. SSE also uses *unsat core* to help backtracking in wrong speculations.

Another research track on boosting symbolic execution focuses on path explosion problem. This research track can be classified into two classes. The first class develops efficient path exploration strategies to achieve specific goals with limited resources, including branch/statement coverage, statement reachability, *etc.* For example, the SGS strategy steers the symbolic execution to less-traveled paths to improve statement coverage [27]. Directed symbolic execution proposes two search strategies to reach a particular target statement [29]. CGDS strategy prioritizes branches with the shortest distance to the unexplored program part, aiming to attain better branch coverage with fewer test inputs [4]. The recently proposed adaptive search heuristic uses machine learning techniques to learn search strategy online to improve the coverage. The learned search strategy is adaptive with respect to the program under test. Experimental results also show that the learned search strategy outperforms traditional fixed search strategy in both statement coverage and bug-finding [7, 9]. Another class of research work reduces the path space so that uninteresting paths are abandoned. Grammar-based white-box fuzzing uses the grammar specification of valid inputs, in order to avoid non-parsable inputs and reach deeper program parts [17]. In [8], the input template is automatically learned online to reduce the path space and hence improve the branch coverage. The works proposed in [39, 47] use program slicing to reduce paths unrelated to the analysis target. For concurrent programs, partial order reduction can be used to reduce path space [37, 43]. State merging techniques also can reduce the number of paths effectively [2, 24]. It worths noting that state merging with the *ite* operator encodes multiple paths into one formula, which is solved by the constraint solver. This also can be seen as using constraint solving to search the path space directly. However, as far as we know, we are the first to open up the constraint solver in symbolic execution.

Search based software testing (SBST) techniques use fitness function to measure how close a test input can reach the target program part, such as branches and statements, and then employ optimization algorithms to find the global minimum of the fitness function [30, 45]. SBST transforms the test input generation problem into a mathematical optimization problem so that plenty of optimization algorithms can be used. In Section 3.4, we discuss the optimization-based constraint solving technique used in search-based testing.

6 CONCLUSION

Symbolic execution is facing the scalability problem caused by the path explosion and the complexity explosion inside the constraint solver. We observe that there exist redundant searchings in the stack of symbolic execution. In this paper, we propose MuSE, a general method to use the constraint solver to explore the path space directly. MuSE maps the search procedure of the constraint solving algorithm to the searchings in the path space via extracting partial solutions produced in the solving procedure. We implement MuSE in mainstream symbolic execution engines and the state-of-the-art constraint solvers. The experimental results show that MuSE achieves one or two orders of magnitude speedups on the three constraint solving methods to reach the same code coverage. We believe that MuSEis the first step towards unifying the searching procedures in symbolic execution. There are several directions for future work: 1) implementing MuSE on more theories and conducting more extensive experiments; 2) investigating the synergy of concrete and symbolic executions to leverage partial solutions more efficiently; 3) exploring the methods for unifying the searching procedures under different backgrounds.

REFERENCES

[1] 2020. LibFuzzer. http://llvm.org/docs/LibFuzzer.html.

[2] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2014. Enhancing Symbolic Execution with Veritesting. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 1083–1094. https://doi.org/10.1145/2568225.2568293

[3] Berkeley. 2020. SoftFloat 2b. http://www.jhauser.us/arithmetic/SoftFloat.html.

[4] J. Burnim and K. Sen. 2008. Heuristics for Scalable Dynamic Test Generation. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. 443–446.

[5] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. 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* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[6] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2008. EXE: Automatically Generating Inputs of Death. *ACM Trans. Inf. Syst. Secur.* 12, 2, Article 10 (Dec. 2008), 38 pages. https://doi.org/10.1145/1455518.1455522

[7] Sooyoung Cha, Seongjoon Hong, Junhee Lee, and Hakjoo Oh. 2018. Automatically Generating Search Heuristics for Concolic Testing. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 1244–1254. https://doi.org/10.1145/3180155.3180166

[8] S. Cha, S. Lee, and H. Oh. 2018. Template-Guided Concolic Testing via Online Learning. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 408–418.

[9] Sooyoung Cha and Hakjoo Oh. 2019. Concolic Testing with Adaptively Changing Search Heuristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 235–245. https://doi.org/10.1145/3338906.3338964

[10] Brett Daniel, Tihomir Gvero, and Darko Marinov. 2010. On Test Repair Using Symbolic Execution. In *Proceedings of the 19th International Symposium on Software Testing and Analysis* (Trento, Italy) *(ISSTA '10)*. Association for Computing Machinery, New York, NY, USA, 207–218. https://doi.org/10.1145/1831708.1831734

[11] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[12] Isil Dillig, Thomas Dillig, and Alex Aiken. 2009. Cuts from Proofs: A Complete and Practical Technique for Solving Linear Inequalities over Integers. In *Computer Aided Verification*, Ahmed Bouajjani and Oded Maler (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 233–247.

[13] Bruno Dutertre and Leonardo de Moura. 2006. A Fast Linear-Arithmetic Solver for DPLL(T). In *Computer Aided Verification*, Thomas Ball and Robert B. Jones (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 81–94.

[14] The Free Software Foundation. 2020. GNU Scientific Library. https://www.gnu.org/software/gsl/.

[15] Zhoulai Fu and Zhendong Su. 2016. XSat: A Fast Floating-Point Satisfiability Solver. In *Proceedings of the 28th International Conference on Computer Aided Verification*. 187–209.

[16] Vijay Ganesh and David L. Dill. 2007. A Decision Procedure for Bit-Vectors and Arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification* (Berlin, Germany) *(CAV'07)*. Springer-Verlag, Berlin, Heidelberg, 519–531.

[17] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-Based Whitebox Fuzzing. *SIGPLAN Not.* 43, 6 (June 2008), 206–215. https://doi.org/10.1145/1379022.1375607

[18] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. *SIGPLAN Not.* 40, 6 (June 2005), 213–223. https://doi.org/10.1145/1064978.1065036

[19] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Queue* 10, 1 (Jan. 2012), 20–27. https://doi.org/10.1145/2090147.2094081

[20] Xiangyang Jia, Carlo Ghezzi, and Shi Ying. 2015. Enhancing Reuse of Constraint Solutions to Improve Symbolic Execution. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 177–187. https://doi.org/10.1145/2771783.2771806

[21] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252

[22] S Kirkpatrick, C D Gelatt, and M P Vecchi. 1983. Optimization by Simulated Annealing. *Science* 220, 4598 (1983), 671–680. https://doi.org/10.1126/science.220.4598.671

[23] Daniel Kroening and Ofer Strichman. 2008. *Decision Procedures: An Algorithmic Point of View*. https://doi.org/10.1007/978-3-540-74105-3

[24] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient State Merging in Symbolic Execution. *SIGPLAN Not.* 47, 6 (June 2012), 193–204. https://doi.org/10.1145/2345156.2254088

[25] Leon S Lasdon, Richard L Fox, and Margery W Ratner. 1963. *Linear Programming and Extensions*. Princeton University.

[26] Eva K Lee and John E Mitchell. 2009. *Integer programming: branch and bound methodsInteger Programming: Branch and Bound Methods*. Springer US, Boston, MA, 1634–1643. https://doi.org/10.1007/978-0-387-74759-0_286

[27] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. 2013. Steering Symbolic Execution to Less Traveled Paths. *SIGPLAN Not.* 48, 10 (Oct. 2013), 19–32. https://doi.org/10.1145/2544173.2509553

[28] Daniel Liew, Cristian Cadar, Alastair F. Donaldson, and J. Ryan Stinnett. 2019. Just Fuzz It: Solving Floating-Point Constraints Using Coverage-Guided Fuzzing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 521–532. https://doi.org/10.1145/3338906.3338921

[29] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. 2011. Directed Symbolic Execution. In *Static Analysis*, Eran Yahav (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 95–111.

[30] Phil McMinn. 2004. Search-Based Software Test Data Generation: A Survey: Research Articles. *Softw. Test. Verif. Reliab.* 14, 2 (June 2004), 105–156.

[31] Aina Niemetz, Mathias Preiner, and Armin Biere. 2014 (published 2015). Boolector 2.0 system description. *Journal on Satisfiability, Boolean Modeling and Computation* 9 (2014 (published 2015)), 53–58.

[32] Brian Olson, Irina Hashmi, Kevin Molloy, and Amarda Shehu. 2012. Basin Hopping as a General and Versatile Optimization Framework for the Characterization of Biological Macromolecules. *Adv. in Artif. Intell.* 2012, Article 3 (Jan. 2012), 1 pages. https://doi.org/10.1155/2012/674832

[33] Corina S. Păsăreanu and Neha Rungta. 2010. Symbolic PathFinder: Symbolic Execution of Java Bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering* (Antwerp, Belgium) *(ASE '10)*. Association for Computing Machinery, New York, NY, USA, 179–180. https://doi.org/10.1145/1858996.1859035

[34] Minghui Quan. 2016. Hotspot symbolic execution of floating-point programs. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 1112–1114. https://doi.org/10.1145/2950290.2983966

[35] David A. Ramos and Dawson Engler. 2015. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *Proceedings of the 24th USENIX Conference on Security Symposium* (Washington, D.C.) *(SEC'15)*. USENIX Association, USA, 49–64.

[36] Anthony Romano. 2014. Practical Floating-Point Tests with Integer Code. In *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings (Lecture Notes in Computer Science)*, Kenneth L. McMillan and Xavier Rival (Eds.), Vol. 8318. Springer, 337–356. https://doi.org/10.1007/978-3-642-54013-4_19

[37] Koushik Sen. 2006. *Scalable Automated Methods for Dynamic Program Analysis*. Ph.D. Dissertation. USA. Advisor(s) Agha, Gul. AAI3242987.

[38] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. *SIGSOFT Softw. Eng. Notes* 30, 5 (Sept. 2005), 263–272. https://doi.org/10.1145/1095430.1081750

[39] Jiří Slab, Jan Strejček, and Marek Trtík. 2012. Checking Properties Described by State Machines: On Synergy of Instrumentation, Slicing, and Symbolic Execution, Vol. 7437. https://doi.org/10.1007/978-3-642-32469-7_14

[40] Matheus Souza, Mateus Borges, Marcelo d'Amorim, and Corina S. Pasareanu. 2011. CORAL: Solving Complex Constraints for Symbolic PathFinder. In *NASA Formal Methods*.

[41] Nikolai Tillmann and Jonathan de Halleux. 2008. Pex–White Box Test Generation for .NET. In *Tests and Proofs*, Bernhard Beckert and Reiner Hähnle (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 134–153.

[42] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. 2012. Green: Reducing, Reusing and Recycling Constraints in Program Analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina) *(FSE '12)*. Association for Computing Machinery, New York, NY, USA, Article 58, 11 pages. https://doi.org/10.1145/2393596.2393665

[43] Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. 2008. Peephole Partial Order Reduction. In *Tools and Algorithms for the Construction and Analysis of Systems*, C R Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 382–396.

[44] Xinyu Wang, Jun Sun, Zhenbang Chen, Peixin Zhang, Jingyi Wang, and Yun Lin. 2018. Towards optimal concolic testing. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 291–302. https://doi.org/10.1145/3180155.3180177

[45] Joachim Wegener, Andre Baresel, and Harmen Sthamer. 2001. Evolutionary test environment for automatic structural testing. *Information and Software Technology* 43, 14 (2001), 841–854. https://doi.org/10.1016/S0950-5849(01)00190-2

[46] H. Yu, Z. Chen, J. Wang, Z. Su, and W. Dong. 2018. Symbolic Verification of Regular Properties. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 871–881.

[47] H. Yu, Z. Chen, J. Wang, Z. Su, and W. Dong. 2018. Symbolic Verification of Regular Properties. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 871–881.

[48] Y. Zhang, Z. Chen, and J. Wang. 2012. Speculative Symbolic Execution. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. 101–110.