https://doi.org/10.1007/s00165-021-00538-3
BCS © 2021
Formal Aspects of Computing

Formal Aspects of Computing

Machine learning steered symbolic execution framework for complex software code

Lei Bu, Yongjuan Liang, Zhunyi Xie, Hong Qian, Yi-Qi Hu, Yang Yu, Xin Chen and Xuandong Li
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, People's Republic of China

Abstract. During program traversing, symbolic execution collects path conditions and feeds them to a constraint solver to obtain feasible solutions. However, complex path conditions, like nonlinear constraints, which widely appear in programs, are hard to be handled efficiently by the existing solvers. In this paper, we adapt the classical symbolic execution framework with a machine learning approach for constraint satisfaction. The approach samples and learns from different solutions to identify potentially feasible area. This sampling-learning style solving can be applied in different class of complex problems easily. Therefore, incorporating this approach, our framework, MLBSE, supports the symbolic execution of not only simple linear path conditions, but also nonlinear arithmetic operations, and even black-box function calls of library methods. Meanwhile, thanks to the theoretical foundation of the machine learning based approach, when the solver fails to solve a path condition, we can have an estimation of the confidence in the satisfiability (ECS) of the problem to give users insights about how the problem is analyzed and whether they could ultimately find a solution. We implement MLBSE on the basis of Symbolic Path Finder (SPF) into a fully automatic Java symbolic execution engine. Users can feed their code to MLBSE directly, which is very convenient to use. To evaluate its performance, 22 real case programs are used as the benchmarks for MLBSE to generate test cases, which involve a total number of 1042 methods that are full of nonlinear operations, floating-point arithmetic as well as native method calls. Experiment results show that the coverage achieved by MLBSE is much higher than the state-of-the-art tools.

Keywords: Symbolic execution, Machine learning, Nonlinear path condition, Constraint solving

1. Introduction

Symbolic execution [BEL75, Kin76, Cla76] is a program analysis technique which was first proposed in the 1970s. This technique is featured by executing programs with symbolic inputs rather than concrete inputs. It maintains a path condition which is a conjunction of symbolic constraints. The path condition is updated whenever the program executes a branch instruction. By solving the collected constraints of the path condition, we can use this technique to generate concrete inputs which can trigger the corresponding path. Therefore, this technique has been widely used in different areas including software testing, analysis and verification [CGK+11, TDH08].

Correspondence to: Lei Bu, E-mail: bulei@nju.edu.cn

Published online: 26 May 2021

However, the existing symbolic execution tools are unable to support real world programs well. The major difficulty is caused by the complexity of the path conditions. Real world programs always contain complicated constraints involving nonlinear computations, mathematical methods (e.g. sin, log) and floating-point symbolic variables. It has been widely recognized that such constraints are huge obstacles for the existing constraint solvers [Dav73, PV09]. Although there are strong SMT solvers such as iSAT [FHT+07] and Z3 [JDM12] which support nonlinear constraints, a recent evaluation [BVLS13] on SMT solvers using GNU Scientific Library [Gou09] has reported that both iSAT and Z3 can only handle a small percentage of constraints collected from the library.

Besides mathematical expressions, software programs may also involve function calls of black-box library methods or even native methods. It is difficult to present such behavior by path conditions as pure mathematical expressions. As a result, it is a great challenge for constraint solvers to solve such path conditions and to generate concrete input values to trigger the specific path [CC84].

As classical constraint solving cannot scale well to such complex programs, intensive investigations have been conducted to solve this problem. First, concolic testing [Sen07, GKS05, PRV11] executes programs with symbolic and concrete inputs. It replaces complicated symbolic terms with concrete values. This simplification facilitates concolic testing to go through the specific path. However, this simplification also decreases the probability to find more desired solutions.

Another kind of approaches for handling such programs with complicated constraints is heuristic search [SBdP11, BAA+12, DA14]. They try to find the solutions that satisfy the constraints with the help of heuristics like genetic algorithms (GA) [GFA13, LGS96], tabu-search [Glo90], etc. These approaches perform well in dealing with some problems. However, there is few guarantee or understanding on the quality of the solutions found by these heuristic algorithms. Moreover, these approaches commonly have many parameters that effect their performance greatly but are hard to be well preset, and are hard to scale to large problems.

In this paper, we present a symbolic execution framework, MLBSE (Machine Learning Based Symbolic Execution), to overcome the limitations discussed above. In MLBSE, we encode all nonlinear numerical operations, mathematical methods (e.g. sin, log), floating-point expressions and library method calls as (uninterpreted) symbolic constraints in the path conditions. The feasibility problem with a set of such kind of path conditions is transformed into a search problem. Then we employ one of the machine learning guided search approaches, derivative-free optimization methods, to tackle our problem. Compared with other methods, derivative-free optimization methods, especially those model-based optimization methods which solve the problem by iteratively sampling solutions from a machine learning model and learning an updated model from the sampled solutions, has advantages in both theoretical foundation and empirical performance. Its optimization performance is well bounded. Therefore, when it fails to find a feasible solution, it is possible to estimate the confidence of the satisfiability from its optimization error under some conditions, which could provide added confidence for the users.

In order to take advantage of this machine learning style constraint solving, unlike classical symbolic execution framework, MLBSE is adapted to support the interaction between the symbolic execution core and the underlying solver/sampler. MLBSE is responsible for validating whether the sampled points are satisfiable and helps the underlying solver to converge as quickly as possible.

Main contributions of this paper are as follows:

- We introduce a new symbolic execution framework, MLBSE, which is driven by machine learning guided constraint solving. As the underlying solver is based on sampling and learning operations, our framework is able to encode all the difficult operations, including nonlinear constraints and the library method calls, as (uninterpreted) symbolic constraints, and thus, can handle a wide range of real programs.
- The machine learning guided search method is further improved using the knowledge of symbolic execution: A light-weight local search is injected to improve the efficiency; an integer absorption trick is introduced as integers are more often encountered in real-world programs; and the method is extended to be able to search in unbounded search space.

```
1  void example(long x, double y) {
2      double z = Long.numberOfLeadingZeros(x); //library method
3          if(z > e^x)
4              if(sin(y/(y+2)) < 0) //non-linear arithmetic operation
5                  assert(false);
6  }
```

Fig. 1. A simple java program can have complicated conditions. To achieve target statement, line 5, the program calls an uninterpreted native method and executes a non-linear arithmetic operation

- We implement MLBSE on the basis of a Java symbolic execution engine, Symbolic PathFinder [PR10, PVB+13]. Intensive case studies are conducted to generate test cases for 22 real case programs, with a total number of 1042 methods that are full of nonlinear operations, floating-point arithmetic and even native method calls. The experimental results prove that our tool outperforms the other state-of-the-art tools in terms of efficiency, quality, and stability. Furthermore, it is worth noting that, unlike most of the existing tools, MLBSE is fully automatic. Users do not have to modify any single line of their code to use MLBSE.
- Last but not least, when failing to find any feasible solution for a path condition, we can provide an estimated confidence of the problem's satisfiability, while existing heuristic search tools report nothing but the failure. We compute the estimated confidence based on the theoretical foundation of the machine learning guided search. This confidence could provide a hint for the user to determine whether to devote more effort or stop in time.

The rest of this paper is organized as follows. The background knowledge of symbolic execution and derivative-free optimization method is introduced in Sect. 2. Section 3 introduces our symbolic execution framework MLBSE and presents how MLBSE encode complex program behaviors as symbolic constraints. Section 4 describes the derivative-free optimization method used for constraint solving and the added extensions. The tool implementation of MLBSE is introduced in Sect 5 and the intensive case studies are introduced in Sect. 6. The related works are discussed in Sect. 7. Section 8 is the conclusion of the paper.

2. Background

2.1. Symbolic execution

Symbolic execution is a program analysis technique which uses symbolic variables rather than concrete values as inputs. A path condition is a conjunction of constraints which the inputs must satisfy to execute the corresponding path. The path condition is updated at each branch point during the symbolic execution. Then, a constraint solver will be called to solve the path condition to see whether it is feasible or not. In another word, the constraint solver will check whether there is an input valuation to trigger this path. Existing symbolic execution tools are able to handle normal path conditions efficiently. However, complicated path conditions which contain non-linear computations is hard to solve by existing tools. Furthermore, it's hard to generate path conditions for programs with third-party library method calls.

In the code of Fig. 1, classical symbolic execution regards arguments x and y as the symbolic variables. Assume we want to find an input that triggers the path along the statements in line 3, 4 and 5. To cover this path, symbolic execution needs to find a solution to satisfy the corresponding *if* statements in line 3 ($z > e^x$) and line 4 ($sin(y/(y+2)) < 0$). However, ($sin(y/(y+2)) < 0$) which contains non-linear operations with the floating-point number y is difficult for most of the existing solvers. Furthermore, z is calculated by function call Long.numberOfLeadingZeros(x). A normal constraint solver has no way to solve such behavior.

To overcome this limitation, we propose a new machine learning guided symbolic execution framework for programs containing complex arithmetic operations and library method calls. It will be introduced in Sect. 3.

2.2. Machine learning guided search via derivative-free optimization

Finding solutions to a set of symbolic path conditions can be formulated as an optimization problem, and optimization methods can be employed. However, as the symbolic path conditions extracted from real-world programs can be very complex and even involve black-box function calls to library methods, many traditional optimization methods such as gradient-based methods are not applicable, since they struggle to handle local optima, non-differential functions and black-box functions.

Derivative-free optimization has gained significant theoretical advances recently, and many of them are model-based, including the optimistic optimization [Mun14], Bayesian optimization [SSW+16], and classification-based optimization [YQH16]. Model-based optimization methods perform the optimization by a common model-sample cycle. Alg. 1 presents a typical framework: starting from the initialization (such as a uniform sampling from the search space), it samples solutions from the model that form the set S, and evaluates the solutions that form the data B; then B is used to update the model, which is expected to find better solutions from the new model; when the total number of samples runs out, the loop breaks and the best-so-far solution is returned.

ALGORITHM 1: Framework of Classification Model-based Derivative-free Optimization

```
Require:
    $f$: Objective function to be minimized;
    $X$: Solution space;
    $M \in \mathbb{N}^+$: Total number of samples;
    $m \in \mathbb{N}^+$: Number of samples per iteration;
Ensure:
1: $t \leftarrow 1$; $init \rightarrow X_1$// initialize
2: loop
3:     $S_t \leftarrow \{x_1, \ldots, x_m\}$ is a set of $m$ samples drawn i.i.d. from $X_t$
4:     $B_t \leftarrow \{(x_1, y_1), \ldots, (x_m, y_m)\}$, where $y_i = f(x_i)$
5:     $\tilde{x} \leftarrow$ the best-so-far solutions from $argmin_{(x^i, y^i) \in B_t} y$
6:     $X_{t+1} \leftarrow Model(B_t)$    // learning from the sampled data
7:     $M \leftarrow M - m$; $t \leftarrow t + 1$
8:     if $M \leq 0$ then break
9: end loop
10: return $\tilde{x}$
```

Note that this framework only needs to evaluate solutions by calculating the objective function value $f(x)$, but requires no other information such as gradients. Thus it can be applied for non-differentiable and black-box functions, as long as solutions can be evaluated. We could apply it to minimize the number of constraints that a solution violates. Since the optimistic optimization is limited in its poor scalability for the high-dimensional search space, and Bayesian optimization is restricted in its high computational cost per iteration, we employ the newly proposed classification-based optimization in this work which has high efficiency in practice and relatively fast convergence rate with default algorithmic parameters.

3. Methodology

The major challenge of applying symbolic execution to real world programs is solving their complex path conditions involving not only (nonlinear) arithmetic operations but also black-box library calls efficiently. To overcome these limitations, we propose a new framework, MLBSE , which is driven by machine learning based solving.

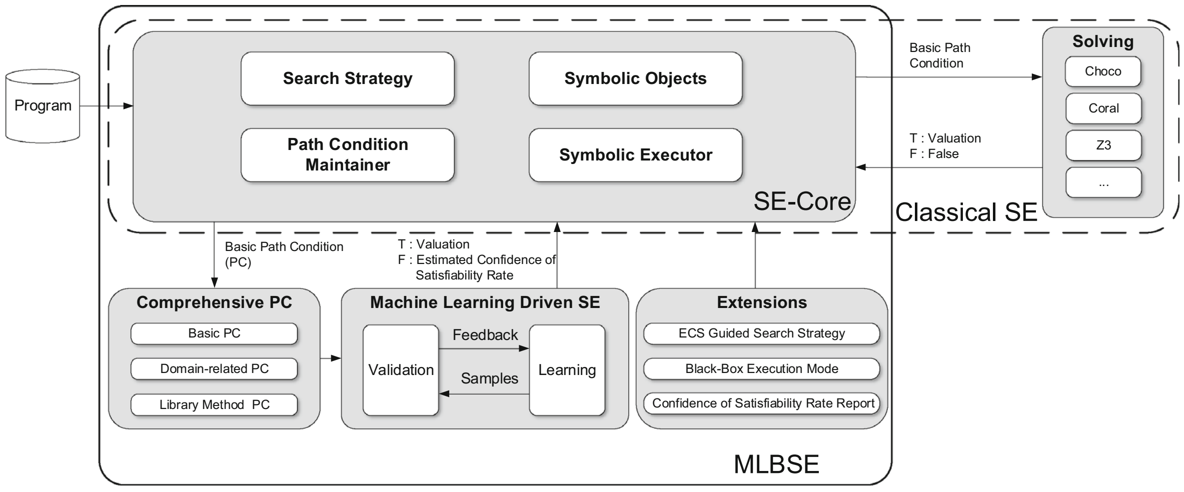

Fig. 2. Architecture of MLBSE

3.1. Architecture

The main process of the classical symbolic execution is divided into two parts: path traversing and path condition solving. The path traversing part executes the target program symbolically and maintains the corresponding path condition. Then the generated path condition is fed to a constraint solver to see whether the corresponding path is feasible or not. As we discussed in Sect. 2.2, if the path condition is complex, it is hard to rely on the underlying solver to give a valuation of the corresponding constraint directly. To solve this problem, a machine learning based symbolic execution framework, MLBSE, is introduced in this paper. MLBSE transforms the feasibility of the path condition into an optimization problem, and then interacts with the machine learning based constraint solving method to find the valuation.

The architecture of MLBSE is depicted in Fig. 2. Besides of the classical symbolic execution core, MLBSE mainly consists of the following three modules:

• Machine learning driven symbolic execution: A significant difference between MLBSE and classical symbolic execution framework is the way to solve the path conditions. In MLBSE, the solving is conducted in a sampling-validation style. A machine learning based underlying solver converges to the correct answer by analyzing the validation feedbacks. In this manner, MLBSE is able to achieve a better efficiency and scalability.

• Comprehensive path condition: During program traversing, symbolic execution collects all the arithmetic conditions in the ongoing path to form the path condition. However, for complex programs which call black box library methods, it is impossible to encode such behavior and to solve it with classical constraint solver. In MLBSE, we encode such function calls as uninterpreted constraints and feed them to the underlying machine learning solver with the basic arithmetic path conditions together. Therefore, MLBSE supports the symbolic execution of a wide range of complex programs.

• Confidence of satisfiability estimation: When classical stochastic search approaches for constraint satisfaction fail to solve the constraints, it is hard to determine if the constraints are satisfiable. The machine learning guided search method employed in MLBSE can help estimate the confidence of satisfiability. When it fails to solve the constraints, we present a strategy to decide if we should devote more resources to solving the problem, using the estimated confidence of satisfiability. The details will be introduced in Sect. 4.

3.2. Machine learning driven SE

MLBSE drives programs' symbolic execution by a new machine learning guided optimization technique reviewed in Sect. 2.2. To apply this technique to our framework, we transform the feasibility problem of the path conditions into an optimization problem through the dissatisfaction function. Varied from most of the existing solvers which solve the path condition directly, our method finds a solution for a path condition by interacting with the symbolic execution engine iteratively (see Fig. 3). The details of this new solving mechanism is discussed in the following section.

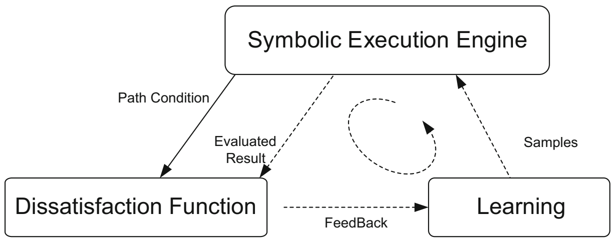

Fig. 3. Workflow of machine learning driven symbolic execution

3.2.1. Satisfaction solving mechanism

To introduce the solving mechanism of path conditions' satisfaction, we first define the dissatisfaction rate for constraints and path conditions below.

Definition 1 (Dissatisfaction rate for constraints) Given a constraint C and a sample S, where C is in the form of 'a op b' (a and b are expressions comprising numbers and variables, $op \in \{>, <, \geq, \leq, =, \neq\}$) and S is a valuation of all variables in C, the dissatisfaction rate of S for C is the distance between S and the theoretical satisfiable sample for the constraint C.

Definition 2 (Dissatisfaction rate for path conditions) Given a path condition PC and a sample S, where PC is a union of constraints and S is a valuation of all variables in PC, the dissatisfaction rate of S for PC is the sum of the dissatisfaction rates for all constraints in S.

The detailed calculation steps of the dissatisfaction rate are given in Sect. 3.2.2. We give some insights on the dissatisfaction rate here. Clearly, the dissatisfaction rate measures how badly a sample violates a path condition, to be specific, a smaller dissatisfaction rate implies better satisfaction of constraints in the path condition. Also, the dissatisfaction rate is always nonnegative and a path condition is satisfied by a sample if and only if the sample's dissatisfaction rate for this path condition is 0.

Thus, our constraint solving mechanism under MLBSE consists of modules as follows:

• Dissatisfaction function. We transform the feasibility problem of the path conditions into a minimum optimization problem. The optimization problem's objective function is a function calculating the dissatisfaction rate for the given path condition. We then take advantage of the machine learning based optimization solving method reviewed in Sect. 2.2.

• Feedback and Learning. Every time the underlying solver proposes a sample for evaluation, the symbolic execution engine evaluates the sample on the path condition by either calculating the arithmetic constraints or executing the library method calls. If not all constraints in the path condition are satisfiable, the dissatisfaction rate is fed back to help the underlying solver converge to the correct solution by machine learning.

3.2.2. Dissatisfaction rate calculation

In this part, we present how to calculate the *dissatisfaction rate* as the metric of feedback. Obviously, if we can let the underlying solver know the distance between the sample and the correct solution more clearly, it can converge to the correct answer more quickly.

Alg. 2 describes how MLBSE calculate the *dissatisfaction rate*.

1. Get the comparator *op* of each constraint C_i in the path condition (line 2);
2. Calculate the distance *d* of C_i with the sample *S* (line 3–5);
3. If the constraint *d op 0* is satisfied, assign 0 to D_i, which means C_i is satisfiable with *S* (line 6–7);
4. if the constraint *d op 0* is not satisfied and the *op* is a "not equal" operation, then assign 1 to *Dist*, otherwise assign $\mid d \mid$ to D_i (line 9–18);
5. One special case is that, if the expression is in the form like $3 < 3$, it is not satisfied, but the value of $d = \mathit{left} - \mathit{right}$ is 0. Therefore, we assign a small epsilon to D_i to indicate the infeasibility (line 12–13).
6. Finally, calculate the dissatisfaction rate D_i for every constraints in the path condition and sum them together as the path condition's dissatisfaction rate.

ALGORITHM 2: Dissatisfaction Rate Calculation

```
Require:
    PC: Path Condition;
    S: Sample in each iteration;
 1: for all $C_i$ in $PC$ do
 2:   $op \leftarrow$ GETCOMPARATOR($C_i$)
 3:   $left \leftarrow$ GETLEFTEXPRVALUE($C_i$,$S$)
 4:   $right \leftarrow$ GETRIGHTEXPRVALUE($C_i$,$S$)
 5:   $d \leftarrow left - right$
 6:   if $d$ $op$ 0 then
 7:     $D_i \leftarrow 0$
 8:   else
 9:     if $op$ is $\neq$ then
10:       $D_i \leftarrow 1$
11:     else
12:       if ($op$ is $>$ $or$ $<$) and ($left == right$) then
13:         $D_i \leftarrow 0.01$
14:       else
15:         $D_i \leftarrow \mid d \mid$
16:       end if
17:     end if
18:   end if
19:   $Sum \leftarrow Sum + D_i$
20: end for
21: return $Sum$
```

3.3. Comprehensive path condition

In addition to the basic arithmetic constraints, the path condition in MLBSE is much more comprehensive. The details of the path conditions enhanced in MLBSE are described as follows.

3.3.1. Domain-related PC

When calculating the path conditions' dissatisfaction rate, samples proposed by the underlying solver may violate the path condition's domain requirement. For example, if the path condition is arcsin(x/y) $<$ 0 and the proposed sample is ($x = 3$, $y = 0$), the calculating process throws a "div by 0" exception.

In order to solve this problem, the domain-related PC is introduced into the path condition in MLBSE. In general, we analyze the constraints in the path condition first to generate extra constraints to check whether the samples satisfy the arithmetic constraint's domain requirements.

In MLBSE, a list of typical arithmetic operations and the related format of domain-related PC is embedded. Before calculating each sample's dissatisfaction rate, the domain-related PC is validated firstly. Only if the domain-related PC is satisfied, we will continue the calculation of the basic path condition's dissatisfaction rate. If not, the validation module abandons the current sample and asks the learning module to re-sample again. For example, for basic path condition (arcsin(x/y) $<$ 0), we generate the domain-related PC($y \neq 0$ && $x/y \leq 1$ && $x/y \geq -1$). Only samples satisfying this domain-related PC before can be fed to the symbolic execution engine to calculate the dissatisfaction rate.

3.3.2. Library method PC

It is easy to find method calls to third-party (black-box) library methods in real world programs. Clearly, we cannot present such library method calls as simple mathematical constraints since the contents of the library methods are uninterpreted. Luckily, in our machine learning style solving, such information is not indispensable. We can present the library method call as an uninterpreted constraint, where the function name is treated as the uninterpreted operator in the constraint and the input arguments of the function call consists of the symbolic variables in this library method PC. For calculating the dissatisfaction rate for the library method PC, we just need to execute this method directly using the arguments in the samples and to compare the actual returned value with the guessed one.

For example, a library method PC ($Double.longBitstoDouble(x) > 0$) is generated and the sample is ($x = 2$). We call the method with argument ($x = 2$) and get the return value 1.16E−123, which is used to calculate the dissatisfaction rate.

3.3.3. Black-box execution mode

As shown in the last section, MLBSE encodes the third-party library method calls as uninterpreted symbolic constraints in the path condition. By this way, we can analyze such programs in a black-box manner. Actually, all the method calls in the program can be processed in such manner . In MLBSE users are given the opportunity to mark specific functions as black box mode in the configuration when they do not want to go through the detail of the specific function. As a result, MLBSE can encode and handle such functions as uninterpreted symbolic constraints directly to increase efficiency.

3.4. Illustration

We illustrate our framework using the program in Fig. 1. Let the assert statement (line 5) be the target statement, and argument x and y be the symbolic variables. To drive the execution to the target statement, we need to find the concrete values of the symbolic variables to satisfy the corresponding path condition. In this example, the path condition maintained by MLBSE is ($Long.numberOfLeadingZeros(x) > e^x$ && $sin(y/(y + 2)) < 0$). Besides we have domain related path condition ($y + 2 \neq 0$).

To solve this path condition, we build the dissatisfaction function through Alg. 2 and calculates the dissatisfaction rate for every sample. Then the optimization model's sampling is guided by minimizing the dissatisfaction rate. For example, one sample proposed by the optimization model is ($x = 13$, $y = -62.83639034682247$), it satisfies the domain-related PC ($y + 2 \neq 0$). For constraint $Long.numberOfLeadin\ Zeros(x) > e^x$, the dissatisfaction rate is calculated as 442353.3920089202. And for constraint $sin(y/(y + 2)) < 0$, the dissatisfaction rate is calculated as 0.858775579161033. Then the path condition's dissatisfaction is 442354.25078449934. The value is fed back to the optimization model to guide the next round sampling. After several rounds of sampling and feeding back, MLBSE gives the final result ($x = 0$, $y = -0.6082275995886306$) and z is calculated as 0 to cover the target line.

4. Framework adaptation and extension

In this section, to facilitate the methodology shown in Sect. 3, we adapt the classification-based algorithm [YQH16] as the derivative-free optimization method in MLBSE and extend it using the domain knowledge of symbolic execution. As an added benefit, the confidence of problems' satisfiability could be provided by the algorithm, guiding a new search strategy for symbolic execution.

4.1. Theoretical analysis of derivative-free optimization via classification

The theoretical analysis of this classification-based optimization framework has been established [YQH16], which discloses that, the learning algorithm in Alg. 1 cannot be arbitrary. Instead, the learning algorithm should have two key quantities: a small *error-target θ-dependence* and a small *γ-shrinkage rate* [YQH16], which implies that the learning algorithm needs to be random and the resulting sampling area should be small.

In detail, it is proved in [YQH16] that, given an objective function f, if a classification-based optimization algorithm has *failure probability* $\delta > 0$ and *approximation level* $\epsilon \geq 0$, the query complexity (i.e., the number of solution samples) is upper bounded by

$$O\Big(\frac{1}{|D_\epsilon|}\Big((1-\lambda) + \frac{\lambda}{\gamma T}\sum_{t=1}^{T}\frac{1 - R_t/(1-\lambda) - \theta}{|D_t|}\Big)^{-1}\ln\frac{1}{\delta}\Big).$$

That is to say, the probability of such an algorithm finds a solution x with additive regret $| f(x) - f^{opt} | \leq \epsilon$ is at least $1 - \delta$, i.e.,

$$P(| f(x) - f^{opt} | \leq \epsilon) \geq 1 - \delta, \tag{1}$$

where f^{opt} is the minimum function value, D_ϵ is the area of the overall target solutions (solutions close to the global optimum with ϵ difference), D_t is the area of the target solutions in iteration t (i.e., the area that $\tilde{D}_t$ approximates), $| \cdot |$ denotes the ratio of the area to the search space, R_t is the prediction error of $\tilde{D}_t$ with respect to D_t, T is the total number of iterations.

4.2. Optimization algorithm adaption and extension

A learning model called randomized coordinate shrinking, named RACOS, was designed and analyzed in [QY16] and [YQH17]. This new machine learning guided search approach method has remarkable merits comparing with other derivative free optimization methods. It is highly efficient in practice and has a fast convergence rate, particularly in high dimensions ranged up to thousands, with default algorithm parameters. Therefore, we adapt RACOS as the derivative-free optimization method in MLBSE and extend it using the domain knowledge of symbolic execution.

The main workflow of our newly extended RACOS is presented in Alg. 3. It adapts the main framework of the classification model-based derivative-free optimization method shown in Alg. 1. In Alg. 3, X denotes the search space with n dimensions, and $I = \{1, ..., n\}$ denotes the dimension index set of X. The objective function to be minimized is denoted as $f_{PC} : X \rightarrow R$, where $f_{PC}(x) = DRC(PC, x)$. Note that DRC is the dissatisfaction rate calculation function defined in Alg. 2 and the parameter PC is described in Sect. 3.3 as the current path condition. A classifier h is a function mapping the search space X to $\{-1, +1\}$ and the classification model $D_h = \{x \in X \mid h(x) = +1\}$ denotes the positive region represented by h. Let $\mathcal{U}_X$ and $\mathcal{U}_{D_h}$ denote the uniform distribution over X and D_h respectively.

The solutions set S is initialized from $\mathcal{U}_X$ randomly (line 2), and then updated iteratively. In each iteration (lines 4–35), it first classifies solutions sampled in the last iteration into positive and negative ones (S^+_{t-1} and S^-_{t-1}) by objective function f (lines 8–9), and then samples m new solutions for current iteration (lines 11–33). To sample new solution x_i, it refines the classification model D_{h_t} by S^+_{t-1} and S^-_{t-1} (lines 12–29) and samples x_i from $\mathcal{U}_{D_{h_t}}$ or $\mathcal{U}_X$ with probabilities (lines 30–32). The classification model D_{h_t}, initialized by a randomly selected positive solution x^+ in S^+_{t-1} (lines 12–14), is then refined in two steps: learning until all negative solutions in S^-_{t-1} are excluded (lines 15–25) and shrinking until there are at most U uncollapsed dimensions (lines 26–29). In the learning step, it selects an uncollapsed dimension index k and a negative solution x^- from S^-_{t-1} randomly (lines 16–17) . To exclude the negative solution x^- from the model D_{h_t}, it sets a bigger lower bound for D_{h_t} in the dimension k , if x^- is smaller than x^+ in this dimension (lines 18–20); or sets a smaller upper bound if x^- is bigger in this dimension (lines 21–23). The best-so-far solution $\tilde{x}$ is updated at the end of each iteration (line 34) and returned if all iterations run out or an optimal one is found (lines 5–7). Since the sampling and learning processes are highly random, it is relatively hard for Alg. 3 to be captured by a local minimum.

In MLBSE, we extend the original derivative-free optimization method to improve efficiency and to support the general unbounded search space. Our main extension includes the injection of a light-weight local search step and the introduction of a reference "solution space" for unbounded search space.

Local search based extension As a machine learning based optimization method, our algorithm generates solutions by sampling. To accelerate its convergence to local minima, we plug in a local search step denoted by function *LocalSearch* in our extended algorithm in Alg. 3 (line 31): for each sampled solution x_i, another solution denoted by x'_i between this solution and the currently selected positive solution x_+ is also sampled, then compare them and return the better one. Note that in this way, the number of samples per iteration is doubled. So that to keep m samples per iteration, only $m/2$ samples should be used in the outer loop.

Unbounded search based extension Also, the original algorithm searches in a bounded solution space, requiring fixed bounds of the search space from its users. The bounds of the search space in our problems, however, are hard to be predetermined in some situations even by human experts. We extend the algorithm to unbounded solution space.

ALGORITHM 3: Extended RACOS in MLBSE

```
Require:
    $f_{PC}$: Objective function to be minimized;
    $\lambda \in (0, 1)$: Balancing parameter (default 0.99);
    $M \in \mathbb{N}^+$: Total number of samples;
    $U \in \mathbb{N}^+$: Maximum number of uncollapsed coordinates;
    $p$: Size of positive solution set;
    $X$: Reference “solution space”;
    $n$: Dimensionality of solution space.
 1: $m = \lfloor\sqrt{M}\rfloor$, and $T = M/m$.
 2: Collect $S_0 = \{x_1, \dots, x_m\}$ by i.i.d. sampling from $\mathcal{U}_X$.
 3: Let $\tilde{x} = argmin_{x \in S_0} f_{PC}(x)$.
 4: for $t = 1$ to $T$ do
 5:     if $f_{PC}(\tilde{x}) = 0$ then
 6:         break
 7:     end if
 8:     $S^+_{t-1}$ = smallest $p$ $x_i$ in $S_{t-1}$ in terms of $f_{PC}$;
 9:     $S^-_{t-1}$ = $S_{t-1}$ - $S^+_{t-1}$;
10:     Let $S_t = \emptyset$
11:     for $i = 1$ to $m$ do
12:         Randomly select $x_+ = (x_+^{(1)}, ..., x_+^{(n)})$ from $S^+_{t-1}$; //Refine space by $x_+$ and $S^-_{t-1}$
13:         $X = ResetBoundary(x_+, X)$
14:         Let $D_{h_t} = X$, $I = \{1, ..., n\}$;
15:         while $\exists\, x \in S^-_{t-1}$ s.t. $h_t(x) = +1$ do
16:             $k$ = randomly selected index from the index set $I$, $I = I - \{k\}$
17:             $x^-$ = randomly selected solution from $S^-_{t-1}$
18:             if $x_+^{(k)} \geq x_-^{(k)}$ then
19:                 $r$ = uniformly sampled value in $(x_-^{(k)}, x_+^{(k)})$
20:                 $D_{h_t} = D_{h_t} - \{x \in X \mid x^{(k)} \leq r\}$
21:             else
22:                 $r$ = uniformly sampled value in $(x_+^{(k)}, x_-^{(k)})$
23:                 $D_{h_t} = D_{h_t} - \{x \in X \mid x^{(k)} \geq r\}$
24:             end if
25:         end while
26:         while $\#I > U$ do
27:             $k$=randomly selected index from $I$,$I = I - \{k\}$
28:             $D_{h_t} = D_{h_t} - \{x \in X \mid x^{(k)} \neq x_+^{(k)}\}$
29:         end while
30:         Sample $x_i$ from $\begin{cases} \mathcal{U}_{D_{h_t}}, & \text{with probability } \lambda \\ \mathcal{U}_X, & \text{with probability } 1-\lambda \end{cases}$
31:         $x_i = LocalSearch(x_i, x_+)$
32:         Let $S_t = S_t \cup \{x_i\}$
33:     end for
34:     $\tilde{x} = argmin_{x \in S_t \cup \{\tilde{x}\}} f_{PC}(x)$
35: end for
36: if $f_{PC}(\tilde{x}) > 0$ then
37:     $level$ = ComputeLevel($f_{PC}$)
38:     $ECS$ = ComputeECS($M$, $n$, $level$)
39: end if
40: return $\tilde{x}$, $f_{PC}(\tilde{x})$ and $ECS$
```

The original algorithm learns a hyper-rectangle model which is initialized by the bounded solution space and shrunk while fitting the data. In our extended version, instead of bounded solution space, a bounding box range called reference "solution space" is input by users. The hyper-rectangle model is then initialized to center at the currently selected positive solution x_+ and has the same size as the reference "solution space", so that its sampling range can be out of the reference "solution space", and can move in unbounded solution space. We denote this process by function $ResetBoundary$ in Alg. 3 (line 13).

4.3. Estimated confidence of satisfiability (ECS)

In previous symbolic execution systems employing stochastic constraint solvers, when a solver reports a path condition is unsolved, we have no idea about how far it is from the satisfaction of the constraint and whether we need to put more resources or not.

Luckily, the theoretical foundation of the machine learning guided search can derive the probability (from Eq.(1)) that a given problem is solvable [YQH16]. This can be helpful for estimating a confidence of satisfiability when we fail to find any feasible solution for symbolic path conditions. The confidence could provide a reference for the user to decide whether more resources should be devoted.

Let F denote the random variable that a feasible solution is found and S denote the random variable that a problem is satisfiable. According to the theory introduced in Sect. 4.1, we have $P(\neg F \mid S) = \delta$, where $P(\neg F \mid S)$ is the failure probability. Also, it is easy to see that $P(\neg F \mid \neg S) = 1$. Since we are interested in the probability that the condition is satisfiable but no feasible solution is found, i.e. $P(S \mid \neg F)$, we compute it as follows.

$$\begin{aligned} P(S \mid \neg F) &= P(\neg F \mid S)P(S)/P(\neg F) && \text{(by Bayes' rule)} \\ &= \frac{P(\neg F \mid S)P(S)}{P(\neg F \mid S)P(S) + P(\neg F \mid \neg S)P(\neg S)} && \text{(by the law of total probability)} \\ &= (1 + \frac{P(\neg S)}{\delta P(S)})^{-1}, && \text{(by } P(\neg F \mid S) = \delta \text{ and } P(\neg F \mid \neg S) = 1) \end{aligned} \tag{2}$$

In Eq. (2), the prior probability of satisfiability $P(S)$ and dissatisfiability $P(\neg S)$ can be estimated from the past experience. Thus, we can provide an estimation of $P(S \mid \neg F)$, which is an estimated confidence about whether the problem is satisfiable when the solver has not found a solution yet.

However, it might be hard for common users to get the unknown values in the equations precisely, e.g., $P(S)$ and $P(\neg S)$. For the ease of use in practice, we replace the calculation of these variables by selecting the problem difficulty levels in our system. We define three preset levels below.

- Easy level: for functions with variable multiplications, whose value changes slowly, e.g., $\sqrt{2} \mid x_1 x_2 \mid^{\frac{1}{2}} = 0$;
- Modest level: for functions with trigonometric expressions, whose value changes fast, e.g., $\sin(2 \mid x \mid) = 0$;
- Hard level: for functions with exponential expressions, whose value changes very fast, e.g., $\mid e^{x+3} - e^3 \mid = 0$.

For each unsolved path condition, we intuitively calculate the difficulty level of its dissatisfaction function as described above by function $ComputeLevel$ in Alg. 3 (line 37). The estimated confidence of satisfiability for the unsolved path condition is then calculated according to Eq. (2) by function $ComputeECS$ in Alg. 3 (line 38). We can roughly set the max value as the whole program's confidence of satisfiability as the confidence of each path is smaller than this one[1]. Considering the difficulty for non-expert users to choose the constraints' difficulty level as required above, the confidence reported is not accurate. It is only used as a hint for the user about whether it is necessary to devote more resources in the analysis of certain problem.

4.4. ECS-guided search strategy

In the classical symbolic execution framework, when the constraint solver fails to solve a path condition or reports a path condition as infeasible, the symbolic execution engine backtracks to check the next branch. However, as MLBSE solves a path condition by the sampling and learning, it is possible that the solver fails to find a solution in the limited rounds when the path condition is feasible.

[1] Definitely, we can get such value for each uncovered branch, instruction and so on in a similar way.

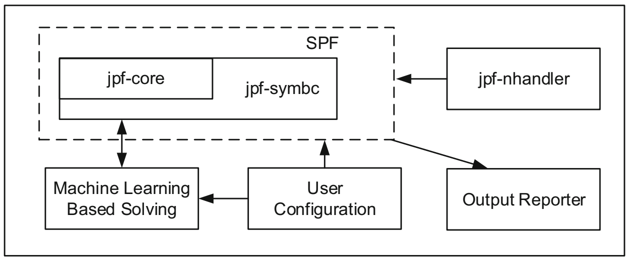

Fig. 4. MLBSE tool architecture

As mentioned in the above paragraph, thanks to the underlying theory-grounded learning solver, we have a estimated confidence of satisfiability for each problem it fails to solve. If the solver is quite confident that the corresponding problem is unsatisfiable, the ECS value it reports will be quite small. Otherwise, we will get a rather large value.

Therefore, by setting an ECS threshold, MLBSE provides an additional search strategy for the users to mitigate the above problem. When MLBSE fails to find a solution for a path condition, if the ECS value of this path condition is above the ECS threshold, MLBSE tries to solve this path condition again with doubled sampling points, until a solution is found or the new ECS value drops below the threshold. Then, MLBSE changes the sample size back to the original setting and continues to solve of the next path. We call this ECS-guided search strategy. Users can specify to use such a strategy during configuration.

5. Tool implementation

The implementation and all the data used in the evaluation are available from GitHub, https://github.com/MLB-SE.

5.1. Tool architecture

MLBSE is implemented on the basis of Symbolic PathFinder (SPF) [PR10, PVB$^+$13], which is a Java symbolic execution engine based on Java PathFinder (JPF) [HP00]. The architecture of MLBSE is shown in Fig. 4.

• The classical program traversing core in SPF provides symbolic path condition generation for the execution semantics. MLBSE adapts it with more comprehensive path condition generation, including the library method call related PC and domain related PC as mentioned in Sect. 3. Meanwhile, it is also enhanced with the ECS-guided search strategy presented in the last section.

• To make MLBSE easier to use, we intergrate jpf-nhandler [SB14] into MLBSE to execute the complex path conditions with native methods more efficiently. The jpf-nhandler module delegates the execution of the native methods as it provides the necessary information of them for MLBSE to generate the uninterpreted constraints.

• To solve the collected path conditions, the classical constraint solving is replaced by a new sampling and validation style machine learning based constraint solving module. A generic interface is created for the frequent interaction between the symbolic execution core and the machine learning based solver. Information required during the sampling and validation iterations is transmitted through the interface so that valid solutions can be quickly found and referred to the exploring process.

• Additionally, we employ and update the convenient publisher system of SPF to provide the program analysis data and produce test cases from the solutions. Users can fetch these data easily and analyze them with the help of other off-the-shelf tools freely. For example, following the merits of [DA14], MLBSE supports the automatic invoking of JaCoCo [jac18] coverage measuring library to demonstrate the coverage reached by the generated test cases.

```
1. target = program                                          //Target class under test
2. classpath = <project-main-class>                          //File path of the target class
3. symbolic.method = program.example(sym#sym)                //Target method under test
4. symbolic.dp = CSP                                         //Default solver setting
5. symbolic.mlpm = 3000                                      //User specified sample size
6. @using = jpf-nhandler                                     //Call jpf-nhandler
7. nhandler.delegateUnhandledNative = true                   //Call jpf-nhandler
8. nhandler.spec.skip = java.lang.Long.numberOfLeadingZeros  //Set the specific func. as black-box mode
```

Fig. 5. Configuration of MLBSE for the example in Fig. 1

5.2. Tool usage

Except the special capability of handling complex programs with complicated constraints and library method calls, another key feature of MLBSE is the user-friendliness. MLBSE is a fully automatic symbolic execution engine for Java. Unlike most of the existing tools, users only need to configure our tool expediently before analyzing their codes. MLBSE does not require users to modify any single line of their codes.

Figure 5 presents an illustration of the configuration for the code shown in Fig. 1 to use MLBSE. As MLBSE is implemented on the basis of SPF, it shares the configuration style of SPF. Line 1–3 in Fig. 5 show the necessary settings required by SPF for declaring the target. It points out the target under test is the method *example* in class *program*.

Line 4 sets the constraint solver under MLBSE to CSP which implements RACOS algorithm [YQH16]. Line 5 is the optional parameter for MLBSE to set the sample size of each round for CSP to 3000. In another word, CSP guesses 3000 valuations each round and presents them to MLBSE to validate. If users do not set the parameter, the default value is 3000. It is worth noting that this is the *only* parameter that relates to the performance of the underlying machine learning solver.

For running the specific function call *java.lang.Long.numberOfLeadingZeros*, line 6–8 are added to call the extension jpf-nhandler. More specifically, line 8 tells MLBSE to encode and analyze the function as an uninterpreted function in a black-box mode.

Furthermore, as mentioned before, MLBSE also provides template script files, following the format of [DA14], to automatically run the test cases and invoke JaCoCo to generates visualized coverage reports. With the help of the script files and the configuration files, users can get the detail coverage report with a push button style user experience. More details can be found on the GitHub link mentioned before.

6. Performance evaluation

6.1. Evaluation setup

To evaluate the performance of MLBSE, intensive sets of experiments using 22 real programs with a total number of 1042 methods are discussed in this section. The first set of experiments introduced in Sect. 6.2 compares the performance of MLBSE with 5 state-of-the-art symbolic execution tools, and evaluates how the configuration parameters influence the performance of MLBSE. Sect. 6.3 discusses the set of experiments to demonstrate MLBSE's special functionality extensions including the black-box execution mode and the estimated confidence of satisfiability report. All the data used and generated in the evaluation are also available from the same GitHub link of the implementation, https://github.com/MLB-SE.

Benchmarks

The programs used in the experiments come mainly from 4 sets of real case benchmarks as shown in Table 1. The first set of programs is from study [DA14, LDG$^+$16]. This set of 13 programs is from real tool distributions and consists of different kinds of computation operations. The second set of the benchmarks are programs selected from the book *Numerical Recipes* [Pre07]. This book introduces sets of classical numerical computation functions widely used in numerous real case programs. The third set of benchmark consists of selected constraint satisfaction problems from [min18]. The fourth set is a real case program *scic* from Github [sci18]. It aims to find numerical solutions for scientific problems and contains of numerical calculation operations.

Table 1. Benchmark for the experiments

Program	Operations	Method	Loc	From
coral	Trigonometric functions, logarithms, polynomials	86	261	[DA14]
dart	Polynomials, required overflow	5	11	[DA14]
hash	Polynomial, shift, bit-wise xor	7	34	[DA14]
opti	Exponentials, square roots	8	29	[DA14]
power	Exponential function	3	20	[DA14]
ray	Polynomials (dot product)	35	190	[DA14]
sine	Float to bit-vector conversion	7	184	[DA14]
stat	Mean and std. dev. computation	17	61	[DA14]
tcas	Constant equality checks	13	82	[DA14, LDG+16]
tsafe	Trigonometric functions	8	65	[DA14, LDG+16]
airy	Polynomials, square roots, logarithms	11	296	[Pre07]
bess	Polynomials, square roots	19	191	[Pre07]
caldat	Polynomials, trigonometric functions	6	86	[Pre07]
ell	Polynomials, trigonometric functions, square roots	31	489	[Pre07]
gam	Logarithms, factorials, exponentials	21	195	[Pre07]
ran	Polynomials, exponentials, xor, logarithms, square roots	13	219	[Pre07]
bound	Polynomials, absolute calculation	21	72	[min18]
color	Polynomials	18	79	[min18]
scic	Polynomials, square roots, exponentials	31	116	[sci18]
mer	Integer and boolean computation	635	2112	[LDG+16]
mine	Boolean computation	44	136	[LDG+16]
wbs	Integer and boolean computation	3	107	[LDG+16]

Competitors

In our experiments, MLBSE and five state-of-the-art symbolic execution tools are used to conduct test case generation on the benchmark. Then, a detailed report containing instruction, branch, line, and cyclomatic complexity [cxt18] coverage is generated by the JaCoCo [jac18] coverage measuring library. Each tool is repeated for 5 times to eliminate the randomness. To prevent small programs from dominating the mean coverage of the tools, we weight each program's contribution with different metrics when computing the arithmetic mean on this specific metric [DA14].

The competitors used in the experiments consist of the following tools: JDart [LDG+16] and jCUTE [SA06] perform concolic execution on Java programs. JDart supports different solvers, like Z3 [JDM12] and CORAL [SBdP11] for complex constraint solving while jCUTE makes all constraints of the path condition remain linear by replacing nonlinear terms of constraints with their concrete run-time value. SPF-Mixed [PRV11], SPF-CORAL [SBdP11] and Concolic Walk (CW) [DA14] are all implemented as an extension of SPF. SPF-Mixed tries to solve nonlinear constraints by getting the solutions of the decidable part and concretely executing the complex part with these solutions. SPF-CORAL and CW solve complex arithmetic path conditions using heuristic search.

All the experiments are conducted on a desktop running Ubuntu 16.04 LTS with 3.40GHz Intel Core i7 and 16GB RAM. The time limit used in the experiment is 300 seconds per method. For example, we give *coral* $86 \times 300 = 25800$ seconds in total.

6.2. Performance study

At the beginning, the coverage of MLBSE is compared on all benchmarks with the state-of-the-art tools mentioned above. The experimental results are summarized in Table 2. The Loc column lists the number of source code lines. The Run time column shows the time each tool took on analyzing the respective program. The Instr. Cov, Branch Cov, Line Cov and Cxty Cov columns show the coverage each tool achieved on the respective program. The Var. columns show the respective variance. The Totals rows denote the total number of instructions, branches, lines and cxtys for each program respectively. To evaluate the effectiveness and efficiency of MLBSE, we come up with two questions as follows:

Table 2. Experiment results

Program	Loc	Tools	Run time (s)	Instr. Cov. (%)		Branch Cov. (%)		Line Cov. (%)		Cxty Cov. (%)	
				Med.	Var.	Med.	Var.	Med.	Var.	Med.	Var.
coral	261	Total		3311		582		261		377	
		JDart	359.61	69%	< 0.01	62%	< 0.01	85%	< 0.01	65%	< 0.01
		jCUTE	585.62	48%	0.01	26%	< 0.01	70%	< 0.01	30%	< 0.01
		SPF-Mixed	141.70	20%	0.01	16%	0.01	39%	< 0.01	18%	< 0.01
		SPF-CORAL	109.19	60%	< 0.01	60%	< 0.01	78%	< 0.01	63%	0.01
		Concolic Walk	46.01	66%	0.05	64%	0.04	78%	0.03	64%	0.03
		MLBSE	62.06	91%	0.02	82%	0.03	91%	< 0.01	80%	0.03
dart	11	Total		42		14		11		12	
		JDart	0.50	71%	< 0.01	86%	< 0.01	82%	< 0.01	67%	< 0.01
		jCUTE	1.40	41%	0.12	24%	0.15	36%	< 0.01	17%	< 0.01
		SPF-Mixed	0.47	57%	< 0.01	50%	< 0.01	64%	< 0.01	42%	< 0.01
		SPF-CORAL	1.35	62%	< 0.01	57%	< 0.01	73%	< 0.01	42%	< 0.01
		Concolic Walk	0.50	71%	< 0.01	85%	0.08	80%	0.13	65%	0.11
		MLBSE	0.49	62%	< 0.01	57%	< 0.01	73%	< 0.01	42%	< 0.01
hash	34	Total		154		10		34		12	
		JDart	2.22	98%	< 0.01	90%	< 0.01	97%	< 0.01	92%	< 0.01
		jCUTE	6.29	94%	< 0.01	70%	< 0.01	91%	< 0.01	75%	< 0.01
		SPF-Mixed	2.22	77%	< 0.01	40%	< 0.01	74%	< 0.01	50%	< 0.01
		SPF-CORAL	2.14	0%	< 0.01	0%	< 0.01	0%	< 0.01	0%	< 0.01
		Concolic Walk	2.20	98%	< 0.01	90%	< 0.01	97%	< 0.01	92%	< 0.01
		MLBSE	2.29	94%	< 0.01	70%	< 0.01	91%	< 0.01	75%	< 0.01
Opti	29	Total		222		32		29		24	
		JDart	3.07	87%	< 0.01	72%	< 0.01	79%	< 0.01	63%	< 0.01
		jCUTE	11.98	61%	< 0.01	41%	< 0.01	67%	< 0.01	42%	< 0.01
		SPF-Mixed	3.08	30%	< 0.01	12%	< 0.01	33%	< 0.01	17%	< 0.01
		SPF-CORAL	8.21	73%	0.06	55%	0.06	79%	< 0.01	58%	0.03
		Concolic Walk	2.91	77%	< 0.01	62%	< 0.01	83%	< 0.01	63%	< 0.01
		MLBSE	5.56	77%	0.19	63%	0.15	70%	0.16	57%	0.04
power	20	Total		56		14		20		10	
		JDart	0.48	88%	< 0.01	100%	< 0.01	85%	< 0.01	80%	< 0.01
		jCUTE	1.99	82%	< 0.01	86%	< 0.01	80%	< 0.01	60%	< 0.01
		SPF-Mixed	0.45	82%	< 0.01	86%	< 0.01	80%	< 0.01	60%	< 0.01
		SPF-CORAL	0.81	88%	< 0.01	100%	< 0.01	85%	< 0.01	80%	< 0.01
		Concolic Walk	0.48	88%	< 0.01	100%	< 0.01	85%	< 0.01	80%	< 0.01
		MLBSE	0.46	88%	< 0.01	100%	< 0.01	85%	< 0.01	80%	< 0.01
ray	190	Total		1405		52		190		61	
		JDart	2.48	63%	< 0.01	50%	< 0.01	75%	< 0.01	44%	< 0.01
		jCUTE	20.56	56%	0.19	60%	0.12	54%	0.13	45%	0.03
		SPF-Mixed	1025.25	34%	< 0.01	23%	< 0.01	44%	< 0.01	25%	< 0.01
		SPF-CORAL	688.47	80%	< 0.01	92%	< 0.01	89%	< 0.01	75%	< 0.01
		Concolic Walk	618.98	78%	< 0.01	90%	< 0.01	88%	< 0.01	74%	< 0.01
		MLBSE	334.83	82%	< 0.01	98%	< 0.01	91%	< 0.01	80%	< 0.01
sine	184	Total		1061		49		184		33	
		JDart	0.47	48%	< 0.01	8%	< 0.01	40%	< 0.01	12%	< 0.01
		jCUTE	4.01	57%	< 0.01	22%	< 0.01	48%	< 0.01	15%	< 0.01
		SPF-Mixed	0.46	0%	< 0.01	0%	< 0.01	0%	< 0.01	0%	< 0.01
		SPF-CORAL	0.47	0%	< 0.01	0%	< 0.01	0%	< 0.01	0%	< 0.01
		Concolic Walk	0.78	86%	< 0.01	53%	0.18	78%	0.02	38%	0.17
		MLBSE	0.73	89%	< 0.01	62%	0.04	82%	< 0.01	53%	0.10
stat	61	Total		200		12		61		23	
		JDart	0.53	66%	< 0.01	67%	< 0.01	57%	< 0.01	35%	< 0.01
		jCUTE	1.70	62%	< 0.01	58%	< 0.01	56%	< 0.01	30%	< 0.01
		SPF-Mixed	0.53	68%	< 0.01	75%	< 0.01	59%	< 0.01	39%	< 0.01
		SPF-CORAL	2.49	69%	< 0.01	83%	< 0.01	61%	< 0.01	43%	< 0.01
		Concolic Walk	0.52	68%	< 0.01	75%	< 0.01	59%	< 0.01	39%	< 0.01
		MLBSE	0.72	68%	< 0.01	78%	0.15	60%	< 0.01	41%	0.05
tcas	82	Total		336		76		82		51	
		JDart	0.69	70%	< 0.01	91%	< 0.01	66%	< 0.01	80%	< 0.01
		jCUTE	9.75	56%	< 0.01	62%	< 0.01	61%	< 0.01	53%	< 0.01
		SPF-Mixed	99.81	70%	< 0.01	91%	< 0.01	63%	< 0.01	80%	< 0.01
		SPF-CORAL	19.96	55%	< 0.01	59%	< 0.01	61%	< 0.01	49%	< 0.01
		Concolic Walk	31.84	70%	< 0.01	91%	< 0.01	63%	< 0.01	80%	< 0.01
		MLBSE	3.05	58%	0.41	62%	1.24	64%	0.12	49%	1.12
tsafe	65	Total		367		26		65		21	
		JDart	0.99	81%	< 0.01	77%	< 0.01	82%	< 0.01	52%	< 0.01
		jCUTE	4.53	78%	0.04	73%	0.26	81%	0.01	48%	0.36
		SPF-Mixed	0.90	0%	< 0.01	0%	< 0.01	0%	< 0.01	0%	< 0.01
		SPF-CORAL	57.04	89%	< 0.01	92%	< 0.01	90%	< 0.01	71%	< 0.01
		Concolic Walk	2.13	89%	< 0.01	92%	< 0.01	90%	< 0.01	71%	< 0.01
		MLBSE	9.78	89%	< 0.01	92%	< 0.01	91%	< 0.01	71%	< 0.01
airy	296	Total		1740		108		296		65	
		JDart	1.21	88%	< 0.01	64%	< 0.01	89%	< 0.01	45%	< 0.01
		jCUTE	12.19	61%	1.17	43%	0.71	63%	1.45	26%	0.06
		SPF-Mixed	0.97	42%	< 0.01	30%	< 0.01	41%	< 0.01	18%	< 0.01
		SPF-CORAL	6.42	44%	< 0.01	31%	< 0.01	45%	< 0.01	20%	< 0.01
		Concolic Walk	1.20	44%	< 0.01	31%	< 0.01	45%	< 0.01	20%	< 0.01
		MLBSE	4.20	97%	< 0.01	82%	< 0.01	98%	< 0.01	69%	< 0.01

Table 2—*cont.* Experiment results

Program	Loc	Tools	Run time (s)	Instr. Cov. (%)		Branch Cov. (%)		Line Cov. (%)		Cxty Cov. (%)	
				Med.	Var.	Med.	Var.	Med.	Var.	Med.	Var.
bess	191	Total		1565		88		191		63	
		JDart	122.15	78%	< 0.01	70%	< 0.01	80%	< 0.01	65%	< 0.01
		jCUTE	104.53	42%	0.87	33%	0.19	32%	0.72	44%	0.30
		SPF-Mixed	3.07	3%	< 0.01	5%	< 0.01	3%	< 0.01	6%	< 0.01
		SPF-CORAL	21.96	16%	0.07	19%	0.07	18%	0.06	17%	0.05
		Concolic Walk	3.65	18%	< 0.01	22%	< 0.01	19%	< 0.01	19%	< 0.01
		MLBSE	8.46	99%	< 0.01	94%	< 0.01	98%	< 0.01	89%	< 0.01
caldat	86	Total		513		48		86		30	
		JDart	50.88	89%	< 0.01	90%	< 0.01	90%	< 0.01	80%	< 0.01
		jCUTE	32.25	93%	1.62	92%	2.46	92%	2.54	83%	1.31
		SPF-Mixed	2.08	56%	< 0.01	46%	< 0.01	64%	< 0.01	33%	< 0.01
		SPF-CORAL	11.58	78%	< 0.01	65%	< 0.01	85%	< 0.01	43%	< 0.01
		Concolic Walk	2.04	62%	< 0.01	52%	< 0.01	66%	< 0.01	40%	< 0.01
		MLBSE	10.28	88%	< 0.01	88%	< 0.01	90%	< 0.01	77%	< 0.01
ell	489	Total		3031		284		489		173	
		JDart	12.57	64%	< 0.01	46%	< 0.01	70%	< 0.01	35%	< 0.01
		jCUTE	141.42	64%	0.16	43%	0.18	68%	0.11	32%	0.08
		SPF-Mixed	625.30	29%	< 0.01	25%	< 0.01	30%	< 0.01	18%	< 0.01
		SPF-CORAL	400.93	50%	< 0.01	24%	< 0.01	47%	< 0.01	22%	< 0.01
		Concolic Walk	671.20	56%	< 0.01	29%	< 0.01	54%	< 0.01	21%	< 0.01
		MLBSE	155.09	90%	< 0.01	87%	< 0.01	91%	< 0.01	79%	< 0.01
gam	195	Total		1240		128		195		85	
		JDart	4.78	86%	< 0.01	77%	< 0.01	85%	< 0.01	64%	< 0.01
		jCUTE	103.02	68%	< 0.01	60%	< 0.01	65%	< 0.01	48%	< 0.01
		SPF-Mixed	305.84	55%	< 0.01	41%	< 0.01	50%	< 0.01	38%	< 0.01
		SPF-CORAL	35.04	64%	< 0.01	48%	< 0.01	58%	< 0.01	42%	< 0.01
		Concolic Walk	6.37	65%	< 0.01	52%	< 0.01	60%	< 0.01	45%	< 0.01
		MLBSE	125.87	92%	< 0.01	80%	< 0.01	93%	< 0.01	66%	< 0.01
ran	219	Total		1181		140		219		83	
		JDart	160.04	96%	< 0.01	75%	< 0.01	98%	< 0.01	55%	< 0.01
		jCUTE	38.57	91%	< 0.01	72%	< 0.01	93%	< 0.01	55%	0.02
		SPF-Mixed	117.95	6%	< 0.01	5%	< 0.01	8%	< 0.01	2%	< 0.01
		SPF-CORAL	1822.35	61%	< 0.01	49%	< 0.01	68%	< 0.01	32%	< 0.01
		Concolic Walk	43.27	10%	< 0.01	6%	< 0.01	13%	< 0.01	5%	< 0.01
		MLBSE	853.08	88%	< 0.01	72%	< 0.01	89%	< 0.01	58%	< 0.01
bound	72	Total		522		106		72		74	
		JDart	33.74	74%	< 0.01	65%	< 0.01	68%	< 0.01	54%	< 0.01
		jCUTE	18.14	71%	< 0.01	69%	< 0.01	69%	< 0.01	61%	< 0.01
		SPF-Mixed	2.29	71%	< 0.01	69%	< 0.01	69%	< 0.01	61%	< 0.01
		SPF-CORAL	6.31	74%	< 0.01	71%	< 0.01	69%	< 0.01	62%	< 0.01
		Concolic Walk	2.30	71%	< 0.01	69%	< 0.01	69%	< 0.01	61%	< 0.01
		MLBSE	2.86	74%	< 0.01	71%	< 0.01	69%	< 0.01	62%	< 0.01
color	79	Total		902		102		79		69	
		JDart	3.39	95%	< 0.01	92%	< 0.01	92%	< 0.01	84%	< 0.01
		jCUTE	28.99	51%	< 0.01	44%	< 0.01	38%	< 0.01	42%	< 0.01
		SPF-Mixed	3.33	81%	< 0.01	82%	< 0.01	74%	< 0.01	74%	< 0.01
		SPF-CORAL	39.57	95%	< 0.01	90%	< 0.01	89%	< 0.01	81%	< 0.01
		Concolic Walk	3.28	81%	< 0.01	82%	< 0.01	74%	< 0.01	74%	< 0.01
		MLBSE	11.79	84%	0.15	76%	0.20	81%	0.05	68%	0.10
scic	116	Total		644		34		116		48	
		JDart	4.22	89%	< 0.01	82%	< 0.01	80%	< 0.01	63%	< 0.01
		jCUTE	10.34	88%	< 0.01	81%	0.09	80%	< 0.01	63%	0.01
		SPF-Mixed	2.74	14%	< 0.01	9%	< 0.01	12%	< 0.01	6%	< 0.01
		SPF-CORAL	21.54	89%	< 0.01	85%	< 0.01	81%	< 0.01	67%	< 0.01
		Concolic Walk	3.98	89%	< 0.01	88%	< 0.01	81%	< 0.01	67%	< 0.01
		MLBSE	9.61	89%	< 0.01	94%	< 0.01	81%	< 0.01	71%	< 0.01
mer	2112	Total		11184		620		2112		945	
		JDart	42.50	81%	< 0.01	51%	< 0.01	77%	< 0.01	64%	< 0.01
		jCUTE	22.74	66%	< 0.01	34%	< 0.01	69%	< 0.01	45%	< 0.01
		SPF-Mixed	23.27	81%	< 0.01	51%	< 0.01	77%	< 0.01	64%	< 0.01
		SPF-CORAL	180.71	81%	< 0.01	51%	< 0.01	77%	< 0.01	64%	< 0.01
		Concolic Walk	23.16	81%	< 0.01	51%	< 0.01	77%	< 0.01	64%	< 0.01
		MLBSE	172.55	81%	< 0.01	51%	< 0.01	77%	< 0.01	64%	< 0.01
mine	136	Total		568		100		136		95	
		JDart	1.52	68%	< 0.01	52%	< 0.01	81%	< 0.01	55%	< 0.01
		jCUTE	4.16	64%	< 0.01	35%	< 0.01	75%	< 0.01	40%	< 0.01
		SPF-Mixed	2.84	68%	< 0.01	52%	< 0.01	81%	< 0.01	55%	< 0.01
		SPF-CORAL	9.01	68%	< 0.01	52%	< 0.01	81%	< 0.01	55%	< 0.01
		Concolic Walk	2.69	68%	< 0.01	52%	< 0.01	81%	< 0.01	55%	< 0.01
		MLBSE	32.20	68%	< 0.01	52%	< 0.01	81%	< 0.01	55%	< 0.01
wbs	107	Total		356		90		107		48	
		JDart	0.49	76%	< 0.01	67%	< 0.01	77%	< 0.01	44%	< 0.01
		jCUTE	9.59	76%	< 0.01	67%	< 0.01	77%	< 0.01	44%	< 0.01
		SPF-Mixed	0.51	76%	< 0.01	67%	< 0.01	77%	< 0.01	44%	< 0.01
		SPF-CORAL	1.11	76%	< 0.01	67%	< 0.01	77%	< 0.01	44%	< 0.01
		Concolic Walk	0.53	76%	< 0.01	67%	< 0.01	77%	< 0.01	44%	< 0.01
		MLBSE	0.79	76%	< 0.01	67%	< 0.01	77%	< 0.01	44%	< 0.01

RQ1: Is MLBSE more effective than other competitors in problem solving of complex programs?

To answer this question, we summarize the data reported in Table 2 on all the programs with respect to different metrics. To prevent small benchmarks dominating the coverage, we compute the arithmetic mean over all the programs, using the respective metric as weights. For example, for line coverage, we use the line of codes as weight, while for branch coverage we use the number of branches as weight. The results are summarized in Fig. 6. In general, MLBSE outperforms all the other competitors significantly on all the coverage metrics. MLBSE's instruction coverage (86%) is about 1.3 times of CW (68%), 1.3 times of SPF-CORAL (65%), 1.7 times of SPF-Mixed (50%) ,1.4 times of jCUTE (63%), and 1.1 times of JDart(77%). MLBSE also outperforms other competitors on branch coverage, line coverage and cyclomatic complexity coverage in a similar scale as well. This set of experiments strengthens our belief that the machine learning based solving technique can benefit symbolic execution of complex nonlinear programs.

Table 2 also displays the detail of performance of MLBSE on each program. We can see that the coverage achieved by MLBSE outperforms other competitors on most of the programs, especially on the nonlinear computation algorithm benchmarks from Numerical Recipes.

RQ2: Is MLBSE more efficient than other competitors?

To evaluate the efficiency of MLBSE, we use the coverage including instruction, branch, line and cyclomatic complexity achieved per unit of generation time as the metrics. Similar with Fig. 6, we compute the average efficiency over all the benchmarks of every tool and summary the results in Fig. 7.

Take the instruction efficiency for example, MLBSE (0.61%/s) is as efficient as CW (0.61%/s), 2.1 times of SPF-CORAL (0.29%/s), 1.8 times of SPF-Mixed (0.33%/s). The reason is that with the help of the underlying machine learning based solving, MLBSE is able to converge to the correct answer quickly and efficiently. The other metrics share the similar results. Although JDart and jCute are more efficient than MLBSE as they execute program by concolic analysis, MLBSE achieves much higher coverage than them. For example, MLBSE achieves 72% branch coverage while JDart achieves 62% and jCUTE achieves 43%. In conclusion, MLBSE is efficient among traditional symbolic execution tools.

RQ3: What influence does the sample size parameter have on the performance of MLBSE?

After the extension of RACOS, shown in Sect. 4.2, MLBSE has only one parameter need to be chosen, i.e., the sample size. For different programs users can set different sample sizes as the budgets of the experiment. To evaluate how the sample size influences MLBSE's performance, experiments with different sample sizes are carried out and the coverage and the running time are summarized in Fig. 8. Figure 8 demonstrates that setting greater sample size can improve the coverage: when the sample size is small, the improvement is clear, and when the sample size is large, the improvement becomes marginal. Meanwhile, the running time of the system increase with the sample size. Therefore, the user can choose the sample size according to the time budget. Note that a larger sample size will also lead to a smaller ECS value.

6.3. Functionality extensions

In this subsection, we discuss the extensions of MLBSE including the ability of treating function calls in a black-box mode and the special capability of reporting the estimated confidence of satisfiability values.

RQ4: Can MLBSE handle programs containing library method calls?

To evaluate MLBSE's capability of handling library method calls and even native methods in a black-box mode, we choose *The Apache Commons Math* [apa18] as the benchmark. *The Apache Commons Math* is a library of lightweight, self-contained mathematics and statistics components which has been used widely.

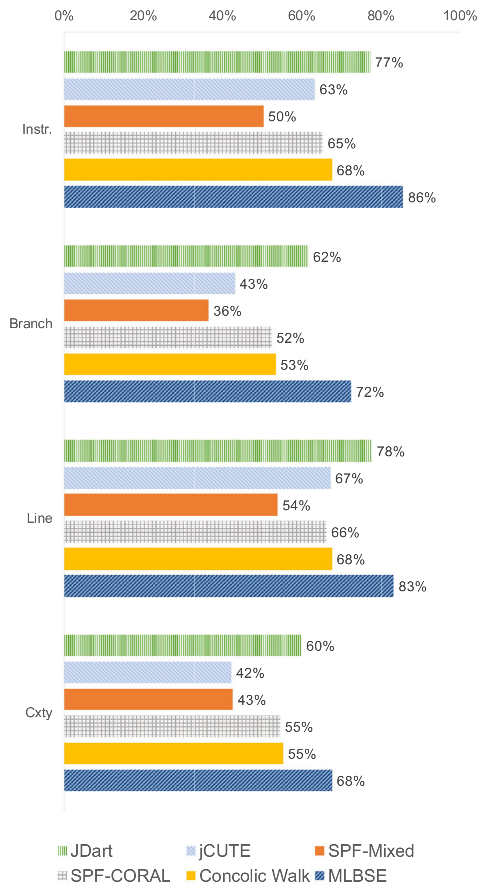

Fig. 6. Weighted coverage reports

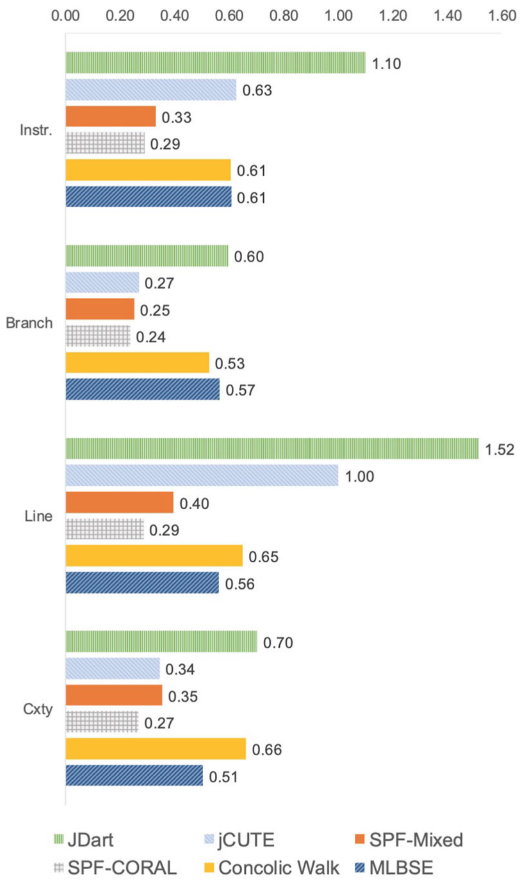

Fig. 7. Weighted efficiency reports

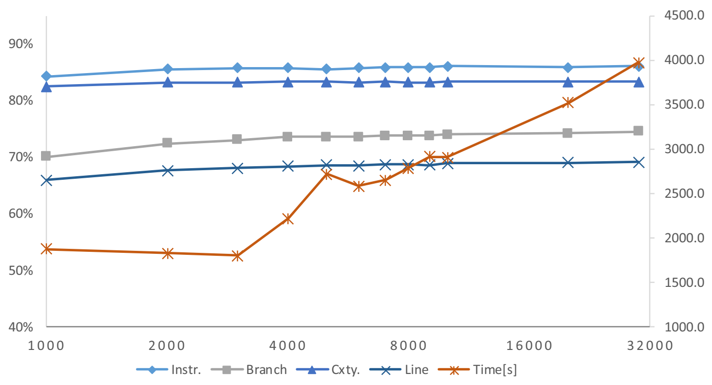

Fig. 8. Influence of the sample size

Table 3. Branch Coverage of Modified *coral*

Tool	Branch Cov (%)	Time (s)
MLBSE-white-box	61	18846.8
MLBSE-black-box	77	222.3
CW-white-box	24	32.1
CW-black-box	7	32.9

Table 4. Estimated Confidence of Satisfiability Report

Program	coral	dart	hash	opti	power	ray	sine	stat	tcas	tsafe	airy
ECS (%)	48	<0.01	33	42	<0.01	49	<0.01	<0.01	42	45	18
Program	bess	caldat	ell	gam	ran	bound	color	scic	mer	mine	wbs
ECS (%)	18	33	39	33	33	39	47	16	49	49	<0.01

We replace the mathematical methods in *coral* with *The Apache Commons Math's* methods. Then, we use MLBSE to generate test cases for the modified *coral* in the black-box mode. As SPF can load the library method calls and go through the instructions step by step, we conduct the classical white-box analysis on this program as well. The branch coverage results are summarized in Table 3.

The data shows that, the black-box mode can achieve a much higher coverage with a much better efficiency. The reason is that in the classical step-by-step white-box mode, we have to go through the complete structure of the methods called. While in the black-box mode, we compact the third-party function calls as uninterpreted symbolic constraints and solve them directly. So, we can put the resource in the traversing of the main program itself to achieve higher coverage.

Besides, CW conducts the symbolic execution in a heuristic searching style. Therefore, they can also support the black-box mode analysis. However, unlike MLBSE which is fully automatic, CW requires that the code is modified according to its regulation to use the black-box mode. We modify the *coral* benchmark according to CW's regulation and also list the performance of CW on this program as well. We find CW only implements the support of a limited class of programs. It is clear to see that MLBSE outperforms CW in all the execution modes.

RQ5: Can MLBSE generate estimated confidence of satisfiability when it fails to solve the path condition?

As we described in Sect. 4.3, in MLBSE we present a new investigation into the report of the confidence of satisfiability when it fails to solve the path condition. We try this method on our benchmarks, the estimated confidence of satisfiability for all the benchmarks are reported in Table 4. This data gives user a hint about whether it is necessary to devote more resources in the analysis of the certain problem. For example, comparing *coral* (48%) and *sine* ($< 0.01\%$), even though the coverage of *coral* is higher than *sine* as shown in Table 2, it is worth spending more resources in analyzing *coral*.

RQ6: Is the ECS value generated reasonable?

In order to validate that the claimed properties of the ECS work in practice, we rerun all the experiments with a larger sample size. Take branch coverage for example, we list the branch coverage and ECS value for each problem under sample size 3000, 30000, and 300000 in Table 5. If we achieve a higher branch coverage or ECS value, it will be marked with symbol $\uparrow$. If the value is lower, it will be marked by $\downarrow$, otherwise, it will be marked as -.

We can see that, after we devote more resources, i.e. sample size, the problems with large ECS value can either achieve a better performance and/or lower the ECS value significantly. For example, the ECS value of *hash* is 33% with sample size 3000, when we increase the sample size from 3000 to 300,000, the branch coverage increases from 70 to 90%, meanwhile, the ECS drops from 33 to $< 0.01\%$.

On the other side, due to the approximation in the calculation of the ECS, we also have cases which have high ECS values, but the branch coverage does not increase along with the sample size. However, we can notice that the ECS values of all these cases drop significantly. The reason is that with small sample, the solver thinks the solution space of the problem is not traversed thoroughly, therefore the solver still holds certain level of confidence about the satisfiability of the problem. When the sample size is increased, the solution space is checked more clearly. Although the solver still fails to find a solution, the ECS confidence drops clearly. In another word, it gives us more confident about the problem is infeasible.

Table 5. Estimated confidence of satisfiability report with different sample size

Sample Size	3000			30000			300000			ECS reasonable?
Program	BC (%)	Time (s)	ECS (%)	BC (%)	Time (s)	ECS (%)	BC (%)	Time (s)	ECS (%)	
coral	82	62.06	48	86 ↑	286.06	37↓	88 ↑	2069.18	< 0.01 ↓	Yes
dart	57	0.49	< 0.01	57 -	0.67	< 0.01 -	57 -	4.03	< 0.01 -	Yes
hash	70	2.29	33	83 ↑	2.91	< 0.01 ↓	90 ↑	5.43	< 0.01 -	**no**
opti	63	5.56	42	66 ↑	15.61	3 ↓	66 -	133.86	< 0.01 ↓	Yes
power	100	0.46	< 0.01	100 -	0.46	< 0.01 -	100 -	0.46	< 0.01 -	Yes
ray	98	334.83	49	98 -	864.87	42 ↓	98 -	7195.42	1 ↓	Yes
sine	62	0.73	< 0.01	62 -	2.99	< 0.01 -	62-	25.48	< 0.01 -	Yes
stat	78	0.72	< 0.01	78 -	1.70	< 0.01 -	78-	11.04	< 0.01 -	Yes
tcas	62	3.05	42	91 ↑	218.30	28 ↓	91 -	2709.25	< 0.01 ↓	Yes
tsafe	92	9.78	45	92 -	120.63	14 ↓	92 -	2194.63	< 0.01 ↓	Yes
airy	82	4.20	18	82 -	27.58	< 0.01 ↓	82 -	234.23	< 0.01 -	Yes
bess	94	8.46	18	94 -	65.99	< 0.01 ↓	94 -	656.47	< 0.01 -	Yes
caldat	88	10.28	33	88 -	95.08	< 0.01 ↓	92 ↑	849.19	< 0.01 -	No
ell	87	155.09	39	87 -	622.37	1 ↓	87 -	6587.55	< 0.01 ↓	Yes
gam	80	125.87	33	79 ↓	389.76	< 0.01 ↓	80 ↑	3606.10	< 0.01 -	Yes
ran	72	853.08	33	72 -	800.25	< 0.01 ↓	72 -	4206.44	< 0.01 -	Yes
bound	71	2.86	39	71 -	4.54	1 ↓	71 -	23.18	< 0.01 ↓	Yes
color	76	11.79	47	85 ↑	76.34	22 ↓	87 ↑	822.45	< 0.01 ↓	Yes
scic	94	9.61	16	97 ↑	36.80	< 0.01 ↓	97 -	1011.70	< 0.01 -	Yes
mer	51	172.55	49	51-	302.74	3 ↓	51-	2905.21	< 0.01 ↓	Yes
mine	52	32.20	49	52 -	38.09	< 0.01 ↓	52 -	605.72	< 0.01 -	Yes
wbs	67	0.79	< 0.01	67-	0.86	< 0.01 -	67 -	1.25	< 0.01 -	Yes
Total	72	1806.75		74 ↑	3974.60		75 ↑	35858.27		90.91%

Another important insight is that, if we compare the data with sample 30,000 and 300,000, we can see that for the problems with ECS smaller than 0.01%, the branch coverage report remains the same for most of the cases. This also validates our declaration that the ECS value could be a good indicator about whether it is worth devoting more resources in the problem solving. Take program *sine* for example, its branch coverage is 62% with sample size 3000, which is quite low, but its ECS value is rather small, $< 0.01\%$. No matter how we increase the sample size, the coverage and ECS value for *sine* remains the same. Clearly, it is worthless to waste more resources on this case.

For a quantitative study, we judge the reasonableness of the ECS on each program as that: the ECS is not reasonable if it is not compatible with the experiment that:

- The ECS value is smaller than 0.01% but increasing the sample size will improve the coverage,
- The ECS value is large but increasing the sample size will not increase the coverage nor decrease the ECS value.

The last column of Table 5 indicates the reasonableness. We can see that only for the cases *hash* and *caldat*, the ECS is lower than 0.01 but increasing the sample size improves the coverage. Meanwhile, we can see that for all the cases, the ECS values are decreased when larger sample sizes are used. Overall, on 90.91% (20/22) of the cases, the ECS is compatible with the experiment and is able to determine if more resources need to be devoted.

RQ7: What is the performance of the ECS-guided strategy?

After we validate the usefulness of the ECS value, we evaluate the performance of the ECS-guided strategy, in another word, double the sample size when the solver fails to solve one path condition and meanwhile the reported ECS value is above the threshold.

We implement the strategy in MLBSE and conduct the experiments in the default sample size 3000 with the ECS threshold 10%. The experiment data is listed in Table 6, we can see that with the help of the ECS-guided strategy, it achieves the highest coverage with not significant overhead.

It is interesting to see that the coverage data achieved by the ECS-guided strategy is even higher than the classical strategy with sample size 300,000. After investigation, we find the reason is that, when a problem is not solved, the ECS-guided strategy could double the sample size for many rounds for the specific problem.

Table 6. Performance of different search strategies

Strategy	ECS-guided	Classical	Classical	Classical
Sample size	3000/10%	3000	30000	300000
Instr. Cov. (%)	**87**	86	86	86
Branch Cov. (%)	**76**	72	74	75
Line Cov. (%)	83	83	83	83
Cxty Cov. (%)	**70**	68	69	**70**
Time (s)	3227.71	1806.75	3974.60	35858.27

In this manner, it tries to solve the same path condition many times with a large sample size and happens to solve some difficult problems due to the randomness behind the learning algorithm.

7. Related work

This paper brings a new machine learning based method into the symbolic execution problem to conduct test case generation. The discussion of the related work is organized in the respective directions.

Symbolic execution and complex constraint solving

Symbolic execution is widely used in many areas of software engineering. There are many mature tools including KLEE [CDE+08], SPF[PR10, PVB+13] , Pex [TDH08] and so on. However, it is well recognized that one of the fundamental open problems [ABC+13] of symbolic execution is that complex path conditions involving nonlinear operations and library method calls can not be solved efficiently by available constraint solvers like Z3 [JDM12].

A mitigation strategy for solving complex path conditions is simplifying such path conditions. Concolic execution [Sen07] is one of the representative approaches. It simplifies complex path conditions with concrete values. JDart [LDG+16] uses the stack value for the initial concrete execution and supports different solvers, like Z3 [JDM12] and CORAL [SBdP11] for complex path constraint. jCUTE [SA06] replaces nonlinear terms with their concrete run-time value. SPF-Mixed [PRV11] simplifies nonlinear path conditions by solving the simple part and using the solution as the concrete values of complex part. Besides of concolic execution, which substitutes the path condition by concrete values, there are also approaches which substitute nonlinear constraints by linear envelopes, e.g. [BVLS13], to abstract the state space and make it solvable by linear constraint solvers. Although, simplification-based symbolic execution tools provide the support for nonlinear path conditions, but such simplification can only support limited cases because of the restriction of the search space.

Different from the classical symbolic/concolic execution engine which may simplify the path condition and depend on the underlying solver to solve the path condition directly, our framework, MLBSE, encodes all the complex behaviors as they are. Then MLBSE interacts with the underlying machine learning based solver to evaluate the sample proposed and guides the solver to converge to the correct answer efficiently.

Search-based symbolic execution and testing

MLBSE conducts the solving in a machine learning guided searching mechanism. There are also related search-based approaches which perform symbolic execution and/or test case generation in a heuristic search style [McM04]. The main idea behind these works is to solve complex path conditions by transforming the feasibility problem of the path condition into a searching problem. The input domain of a program is encoded into a searching area and an objective function is used to conduct the searching.

Many efficient search-based solvings like XSat Solver [FS16],CORAL solver [SBdP11] and CW algorithm [DA14] have been proposed in recent works. The XSat solver applies the Metropolis-Hasting [CG95] algorithm while the CORAL solver uses Particle-Swarm Optimization [GRs04] and Alternating Variable Method to solve nonlinear path conditions. The CW algorithm [DA14] splits the path condition into linear and nonlinear parts, then it uses the taboo search to solve the nonlinear part within the polytope induced by the linear part.

However, these heuristic search algorithms are commonly weak in their theoretical foundation. Moreover, their performance is usually very sensitive to the parameters, which are hard to be well preset. The classification-based optimizaition algorithm adapted in MLBSE is a theoretically grounded search algorithm, which not only has a good and stable performance but can also provide the estimated false negative rates in the case that the constraints are not solved.

Last but not least, on the implementation level, like most of the existing tools, [DA14] requires the user to modify their code according to certain regulations firstly for handling library methods. MLBSE is a fully automatic Java symbolic execution engine. Users can feed their code to MLBSE directly, which is very convenient to use. The implementation of MLBSE is first reported in [LLQ+16] in a new idea nature.

8. Conclusions

Complicated path conditions containing nonlinear constraints and calls of library methods appear widely in real world programs. However, limited by the processing capability of the existing constraint solvers, symbolic execution tools fail to analyze such programs efficiently.

To overcome this limitation, we propose a new symbolic execution framework supported by a machine learning based optimization solving technique. The new framework, MLBSE, supports not only linear path conditions, but also nonlinear operations, and even black-box library methods. Meanwhile, MLBSE provides a special functionality to have an estimation of the confidence of the satisfiability (ECS) of a path, which the solver fails to solve, to give users more insights about whether they should devote more resources and could ultimately find a solution.

To evaluate the performance of MLBSE, we generate test cases for 22 real case programs by MLBSE and compare the coverage and the performance of MLBSE with other state-of-the-art tools. The experiment results show that with the help of the interaction with the machine learning style solving, MLBSE supports a wide range of well-known difficult real programs with outstanding efficiency, quality and stability.

Acknowledgements

The authors want to thank the anonymous reviewers and editors for their valuable advices on improving this paper. The authors would also thank Mr. Xin Li, Mr. Yuchao Duan, and Mr. Bochuan Chen for their efforts devoted in developing MLBSE. This work is supported in part by the National Key Research and Development Program of China (2020AAA0107200), the National Natural Science Foundation of China (Nos. 61632015, 61690204, 61876077), and the Leading-edge Technology Prohgram of Jiangsu Natural Science Fundation (No. BK20202001).

References

[ABC+13] Saswat A, Burke Edmund K, Yueh CT, John C, Cohen Myra B, Wolfgang G, Mark H, Jean HM, Phil MM et al (2013) An orchestrated survey of methodologies for automated software test case generation. J Syst Softw 86(8):1978–2001

[apa18] Apache Commons Math (2018) https://commons.apache.org/

[BAA+12] Borges M, Amorim MD, Anand S, Bushnell D, Păsăreanu CS (2012) Symbolic execution with interval solving and meta-heuristic search. In: 2012 IEEE fifth international conference on software testing, verification and validation (ICST). IEEE, pp 111–120

[BEL75] Boyer Robert S, Bernard E, Levitt Karl N (1975) Select—a formal system for testing and debugging programs by symbolic execution. ACM SigPlan Not 10(6):234–245

[BVLS13] Barr Earl T, Thanh V, Le V, Zhendong S (2013) Automatic detection of floating-point exceptions. ACM SIGPLAN Not 48(1):549–560

[CC84] Chang David D, Clayton David A (1984) Precise identification of individual promoters for transcription of each strand of human mitochondrial DNA. Cell 36(3):635–643

[CDE+08] Cristian C, Daniel D, Engler Dawson R et al (2008) Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. OSDI 8:209–224

[CG95] Siddhartha C, Edward G (1995) Understanding the metropolis-hastings algorithm. Am Stat 49(4):327–335

[CGK+11] Cadar C, Godefroid P, Khurshid S, Păsăreanu CS, Sen K, Tillmann N, Visser W (2011) Symbolic execution for software testing in practice: preliminary assessment. In: Proceedings of the 33rd international conference on software engineering. ACM, pp 1066–1071

[Cla76] Clarke Lori A (1976) A system to generate test data and symbolically execute programs. IEEE Trans Softw Eng 3:215–222
[cxt18] Cyclomatic Complexity (2018) http://eclemma.org/jacoco/trunk/doc/counters.html
[DA14] Dinges P, Agha G (2014) Solving complex path conditions through heuristic search on induced polytopes. In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering. ACM, pp 425–436
[Dav73] Martin D (1973) Hilbert's tenth problem is unsolvable. Am Math Mon 80(3):233–269
[FHT+07] Martin F, Christian H, Tino T, Stefan R, Tobias S (2007) Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. J Satisf Boolean Model Comput 1:209–236
[FS16] Fu Zhoulai S, Zhendong, (2016) Xsat: a fast floating-point satisfiability solver. In: Chaudhuri S, Farzan A (eds) Computer aided verification. Springer, Cham, pp 187–209
[GFA13] Galeotti JP, Fraser G, Arcuri A (2013) Improving search-based test suite generation with dynamic symbolic execution. In: 2013 IEEE 24th international symposium on software reliability engineering (ISSRE). IEEE, pp 360–369
[GKS05] Patrice G, Nils K, Koushik S (2005) Dart: directed automated random testing. ACM Sigplan Not 40(6):213–223
[Glo90] Fred G (1990) Tabu search: a tutorial. Interfaces 20(4):74–94
[Gou09] Gough B (2009) GNU scientific library reference manual. Network Theory Ltd, Surrey
[GRs04] Gies D, Rahmat-samii Y (2004) Particle swarm optimization (pso) for reflector antenna shaping. In: Antennas and propagation society international symposium, 2004. IEEE, vol 3, pp 2289–2292
[HP00] Klaus H, Thomas P (2000) Model checking java programs using java pathfinder. Int J Softw Tools Technol Transf 2(4):366–381
[jac18] Jacoco (2018) http://www.eclemma.org/jacoco/
[JDM12] Jovanović D, De Moura L (2012) Solving non-linear arithmetic. In: Gramlich B, Miller D, Sattler U (eds) Automated reasoning. Springer, Berlin, pp 339–354
[Kin76] Kingl James C (1976) Symbolic execution and program testing. Commun ACM 19(7):385–394
[LDG+16] Luckow K, Dimjašević M, Giannakopoulou D, Howar F, Isberner M, Kahsai T, Rakamarić Z, Raman V (2016) JDart: a dynamic symbolic analysis framework. In: Chechik M, Raskin J-F (eds) Proceedings of the 22nd international conference on tools and algorithms for the construction and analysis of systems (TACAS), lecture notes in computer science, vol 9636. Springer, Berlin, pp 442–459
[LGS96] Willisa L, Geuze Hans J, Slot Jan W (1996) Improving structural integrity of cryosections for immunogold labeling. Histochem Cell Biol 106(1):41–58
[LLQ+16] Li X, Liang Y, Qian H, Hu Y-Q, Bu L, Yu Y, Chen X, Li X (2016) Symbolic execution of complex program driven by machine learning based constraint solving. In: Lo D, Apel S, Khurshid S (eds) Proceedings of the 31st IEEE/ACM international conference on automated software engineering, ASE 2016, Singapore, September 3–7, 2016. ACM, pp 554–559
[McM04] Phil MM (2004) Search-based software test data generation: a survey. Softw Test Verif Reliab 14(2):105–156
[min18] Minizinc (2018) http://www.minizinc.org/
[Mun14] Munos R (2014) From bandits to Monte-Carlo tree search: the optimistic principle applied to optimization and planning. Found Trends Mach Learn 7(1):1–130
[PR10] Păsăreanu CS, Rungta N (2010) Symbolic pathfinder: symbolic execution of java bytecode. In: Proceedings of the IEEE/ACM international conference on automated software engineering. ACM, pp 179–180
[Pre07] Press William H (2007) Numerical recipes: the art of scientific computing, 3rd edn. Cambridge University Press, Cambridge
[PRV11] Păsăreanu CS, Rungta N, Visser W (2011) Symbolic execution with mixed concrete-symbolic solving. In: Proceedings of the 2011 international symposium on software testing and analysis. ACM, pp 34–44
[PV09] Păsăreanu Corina S, Willem V (2009) A survey of new trends in symbolic execution for software testing and analysis. Int J Softw Tools Technol Transf 11(4):339–353
[PVB+13] Păsăreanu Corina S, Willem V, David B, Jaco G, Peter M, Neha R (2013) Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. Autom Softw Eng 20(3):391–425
[QY16] Qian H, Yu Y (2016) On sampling-and-classification optimization in discrete domains. In: Proceedings of the 2016 IEEE congress on evolutionary computation (CEC'16), Vancouver, Canada, pp 4374–4381
[SA06] Sen K, Agha G (2006) Cute and jcute: concolic unit testing and explicit path model-checking tools. In: Computer aided verification. Springer, Berlin, pp 419–423
[SB14] Shafiei N, van Breugel F (2014) Automatic handling of native methods in java pathfinder. In: Proceedings of the 2014 international SPIN symposium on model checking of software. ACM, pp 97–100
[SBdP11] Souza M, Borges M, d'Amorim M, Păsăreanu CS (2011) Coral: solving complex constraints for symbolic pathfinder. In: NASA formal methods. Springer, Berlin, pp 359–374
[sci18] Scientific Computation (2018) https://github.com/elizabethzhenliu/ScientificComputation
[Sen07] Sen K (2007) Concolic testing. In: Proceedings of the twenty-second IEEE/ACM international conference on automated software engineering. ACM, pp 571–572
[SSW+16] Bobak S, Kevin S, Ziyu W, Adams Ryan P, de Freitas Nando, (2016) Taking the human out of the loop: a review of Bayesian optimization. Proc IEEE 104(1):148–175
[TDH08] Tillmann N, De Halleux J (2008) Pex–white box test generation for. net. In: Tests and proofs. Springer, Berlin, pp 134–153
[YQH16] Yu Y, Qian H, Hu Y-Q (2016) Derivative-free optimization via classification. In: Proceedings of the 30th AAAI conference on artificial intelligence (AAAI'16), Phoenix, AZ
[YQH17] Yu Y, Hu Y-Q, Qian H (2017) Sequential classification-based optimization for direct policy search. In: Proceedings of the 31st AAAI conference on artificial intelligence (AAAI'17), San Francisco, CA, pp 2029–2035

Received 10 September 2020
Accepted in revised form 30 January 2021 by Zhiming Liu, Xiaoping Chen, Ji Wang and Jim Woodcock