Selective Concolic Testing

Guofeng Zhang[1], Zhenbang Chen[1(✉)], Ziqi Shuai[1], Jun Sun[2], Weijiang Hong[1], Yufeng Zhang[3], Ji Wang[1], and Yang Liu[1]

[1] College of Computer Science and Technology, National University of Defense Technology, Changsha, China
{zhangguofeng16,zbchen,szq,hongweijiang17,wj,liuyang19d}@nudt.edu.cn
[2] Singapore Management University, Singapore, Singapore
junsun@smu.edu.sg
[3] College of Computer Science and Electronic Engineering, Hunan University, Changsha, China
yufengzhang@hnu.edu.cn

Abstract. The principled combination of symbolic execution and random testing lacks a formal foundation, especially in deciding which inputs to symbolize. We propose selective concolic testing, a cost-aware framework that formulates this choice as an optimized policy problem of a MDP (Markov Decision Process). We model program exploration over a finite control-flow graph, where MDP states represent covered statements, actions partition path constraints into symbolic and random fragments, rewards reflect coverage gain, and costs account for SMT solving effort and sampling inefficiency. Our framework yields the first formal characterization of selective symbolization as policy synthesis in a probabilistic system. We prove that exact policy computation is intractable due to the exponential state space and the hardness of solution-density estimation via model counting. Our formulation enables a practical approximation: we partition constraint dependency graphs and use machine learning to predict solver timeouts, guiding per-constraint symbolization decisions. Built on top of KLEE and JFS, our prototype validates the approach on real-world floating-point benchmarks. Results show that selectively symbolizing inputs, guided by predicted solvability and cost, significantly improves coverage efficiency. Our work thus provides both a rigorous theoretical foundation and a practical instantiation for hybrid program analysis.

Keywords: Concolic Testing · Symbolic Execution · MDP · Constraint Solving

1 Introduction

Symbolic execution [8,31] provides a semantically grounded method for precise program analysis, enabling automated test generation [10,12], bug detection [11],

G. Zhang, Z. Chen, Z. Shuai, W. Hong, J. Wang, and Y. Liu—Also with the affiliation: State Key Laboratory of Complex & Critical Software Environment, National University of Defense Technology, Changsha, China

A. Sampaio and M. Stoelinga (Eds.): FM 2026, LNCS 16557, pp. 275–295, 2026.
https://doi.org/10.1007/978-3-032-26220-2_14

and bounded verification [20]. Despite its success in tools such as SAGE [23], PEX [41], and MAYHEM [14], scalability remains limited by path explosion and the intrinsic complexity of constraint solving [13].

Concolic testing [22,38] addresses these limitations by interleaving concrete and symbolic execution: it starts with a concrete input, collects path constraints symbolically, and iteratively generates new inputs via constraint solving to explore new paths. Conceptually, it combines random testing (for initial/-divergent inputs) and constraint-based symbolic testing (for systematic exploration). Random testing is cheap but struggles with low-probability conditions (e.g., magic numbers); symbolic testing is effective for path coverage but incurs high solving overhead, especially for non-linear constraints. An optimal combination would maximize efficiency, yet computing such a strategy is challenging due to its complexity [44]. Prior work optimizes this trade-off for specific goals like statement [44] or branch coverage [9], but assumes full symbolization of all inputs.

We argue that the *policy of input symbolization, i.e.*, which inputs to treat symbolically, is equally critical. Different symbolization choices affect both the feasibility of random testing (via solution probability) and the cost of constraint solving. Thus, selectively symbolizing inputs can outperform approaches that uniformly apply constraint solving to all variables.

To this end, we propose *selective concolic testing*, which formalizes optimal input symbolization as the optimal policy problem of a cost-aware Markov Decision Process (MDP) with respect to statement coverage. For the first time, we cast the problem as a partitioning task based on SMT counting [18] and solving cost. Given the intractability of computing the exact optimal policy, we design an approximate algorithm that combines graph partitioning and machine learning to predict the symbolization decisions of constraints, trading minimal prediction overhead for significant efficiency gains. Implemented on KLEE [10] and JFS [33], our approach is evaluated on two benchmark suites. On GSL, it improves statement coverage by 36.89% over the state-of-the-art [44] and achieves 12.08× speedup for equivalent statement coverage.

Our main contributions are:

- **A formal MDP-based model** for selective concolic testing, grounded in program semantics and cost-aware policy synthesis.
- **An approximation algorithm** that translates the intractable optimal policy problem into a tractable constraint-partitioning task using machine learning.
- **An implementation and evaluation** that the framework aligns with empirical efficiency, validating its practical relevance without compromising theoretical rigor.

2 Preliminaries

This section uses an example program to motivate selective concolic testing. Figure 1 shows a program (denoted by $\mathcal{P}$) that accepts two input variables `x` and

```
int foo(double x,int y) {
    int z = 0;
    if (x^7 >= 0) z = z+1;
    if (y == 0) z = z+1;
    return z;
}
```

Fig. 1. An example program.

y, where `x^7` means `power(x,7)`. $\mathcal{P}$ has four paths in total. We first show how concolic testing works on $\mathcal{P}$. Then, we show how selective concolic testing works on $\mathcal{P}$.

2.1 Concolic Testing

Starting from input $\{x \mapsto 0, y \mapsto 1\}$ with both variables symbolized, concolic testing executes $\mathcal{P}$ along path condition $PC_1 : x^7 \geq 0 \wedge y \neq 0$. Using DFS, it next targets $PC_2 : x^7 \geq 0 \wedge y = 0$. Solving PC_2 with an SMT solver (e.g., Z3 [37]) may succeed (e.g., yielding $\{x \mapsto 0, y \mapsto 0\}$), but often times out due to the hardness of floating-point (QF_BVFP) constraints, preventing coverage of this path.

If PC_2 is solved, the next target under DFS is $PC_3 : x^7 < 0$, which is similarly expensive. A timeout here halts exploration; otherwise, assuming solution $\{x \mapsto -1\}$, a random value for y (e.g., -1) yields input $\{x \mapsto -1, y \mapsto -1\}$, covering $PC_4 : x^7 < 0 \wedge y \neq 0$. The final target is $PC_5 : x^7 < 0 \wedge y = 0$, which faces the same solvability issues as PC_2.

In theory, a perfect solver would generate three new inputs to cover all paths. In practice, with a 30-second timeout (on a 3.5GHz CPU), KLEE explores only one path; without timeout, full coverage takes 219 s. Pure random testing fares poorly: while $x^7 \geq 0$ holds with probability 0.5, hitting $y = 0$ occurs with probability 2^{-32}. Thus, using random generation for x and constraint solving for y is preferable, which motivates selective concolic testing.

2.2 Selective Concolic Testing

Selective concolic testing optimally combines random generation and constraint solving by partitioning each target path condition PC_n into two parts: $PC_n = PC_n^r \wedge PC_n^c$. It uses random generation for variables in PC_n^r and constraint solving for those in PC_n^c.

For $PC_2 = (x^7 \geq 0) \wedge (y = 0)$, we set $PC_2^r = x^7 \geq 0$ and $PC_2^c = y = 0$. Since $y = 0$ is trivial for SMT solvers and $x^7 \geq 0$ holds with high probability (0.5), this partition enables fast input generation.

Ideally, PC_n^r should have high solution probability and high solving cost, while PC_n^c should have low cost and low probability, which is formally defined as an optimization problem in Sect. 3. Computing exact solution probabilities requires model counting [18], which is intractable. Hence, we propose a

lightweight approximation in Sect. 4. For the program in Fig. 1, our implementation completes full path exploration in one second.

3 MDP Theoretical Framework

This section gives a Markov Decision Process (MDP) based theoretical framework for defining the optimized combination of random generation and constraint solving with the selective input variable symbolization.

Definition 1. *A program $\mathcal{P}$ is a tuple (I, N, E), where I represents the program's input variable set, N is the node set of program statements, and $E \subseteq N \times N$ is the edge set representing $\mathcal{P}$'s control flows. Assume the sets of inputs I and non-deterministic choices N are finite.*

Let $\mathsf{pre}(n)$ and $\mathsf{succ}(n)$ denote the predecessor and successor node sets of n. Without loss of generality, assume $\mathcal{P}$ has a single entry node $entry(\mathcal{P})$. A *path* p is a node sequence $\langle n_0, \ldots, n_k \rangle$ with $(n_i, n_{i+1}) \in E$ for $0 \leq i < k$. We write $\mathcal{N}(p)$ for the nodes in p, and $path(n)$ for all paths from $entry(\mathcal{P})$ to n. Note that $path(n)$ may be infinite if $\mathcal{P}$ contains loops. Let $path(\mathcal{P})$ be the set of all program paths, and $PC(p)$ the path condition of p, computed via concolic execution along p. For input i, $\mathcal{P}(i)$ denotes the execution path of $\mathcal{P}$ on i, and by assumption, every node $n \in N$ is reachable: $\forall n \in N$, $\exists i$ s.t. $\mathcal{P}(i) \in path(n)$.

Definition 2 ***(Assignment).*** *Given a variable set V and an SMT theory whose domain is $\mathcal{D}$, an assignment $\mathbb{S} : V \rightarrow \mathcal{D}$ is a function that assigns V's each variable a value in $\mathcal{D}$.*

Definition 3 ***(Solution Space).*** *Given an SMT formula φ, the solution space, denoted as $\Omega(\varphi)$, constitutes all assignments satisfying φ.*

If φ is unsatisfiable, $\Omega(\varphi)$ is $\emptyset$. $\Omega(true)$ is the set of any assignments of $V(\varphi)$, where $V(\varphi)$ represents the set of φ's variables. If the underlying SMT theory has an infinite domain, *i.e.*, `QF_LIA` [5], $\Omega(\varphi)$ may be infinite, *e.g.*, $\Omega(x \geq 0)$, where $x \geq 0$ is a linear integer arithmetic formula and x is an integer variable. In symbolic execution, our analysis is restricted to bit-vector theories with bounded width (e.g., 32-bit or 64-bit), ensuring that $\Omega(\varphi)$ does not represent an infinite domain.

Definition 4 ***(Solution Ratio).*** *Given an SMT formula φ, the solution ratio is the rate of all the solutions that satisfy φ in the whole assignment space. We use $Pr(\varphi)$ to denote φ's solution ratio. $Pr(\varphi) = \frac{\#\Omega(\varphi)}{\#\Omega(true)}$, where $\#S$ represents set S's cardinality.*

For example, if $x \geq 0$ is a bit-vector SMT formula and x is a 32-bit signed integer variable, $Pr(x \geq 0)$ is 0.5. In principle, calculating the solution ratio is a problem of model counting [18]. **In this paper, we only consider the programs whose variables are machine variables, which can be precisely represented by bit-vector based SMT theories.**

Let $\mathbb{R}$ be the set of real numbers. Then, we first define Markov Chain.

Definition 5 ***(MC).*** *A Markov Chain (MC) is a tuple* $(\mathcal{S}, \delta, \mathcal{I})$*, where*

- $\mathcal{S}$ *is a finite set of states.*
- $\delta : \mathcal{S} \times \mathcal{S} \to \mathbb{R}$ *is the transition probability matrix and satisfies the following condition:* $\sum_{u \in \mathcal{S}} \delta(s, u) = 1$ *for each state* s *in* $\mathcal{S}$*.*
- $\mathcal{I} : \mathcal{S} \to \mathbb{R}$ *is the initial distribution of* $\mathcal{S}$ *such that* $\sum_{s \in \mathcal{S}} \mathcal{I}(s) = 1$*.*

Because an MC has a similar structure as a program, we use the similar path notations of programs for Markov Chains. We use $Pr(p, \mathcal{M})$ to represent the probability of the path $p = \langle s_0, .., s_n \rangle$ starting from s_0 in an MC $\mathcal{M} = (\mathcal{S}, \delta, \mathcal{I})$, and $Pr(p, \mathcal{M}) = \Pi_{0 \leq i \leq n-1} \delta(s_i, s_{i+1})$. Besides, we use $\mathsf{Start}(s)$ to represent the set of paths that starts from state s and ends at a sink state s'. For state s, let $\mathsf{succ}(s') = \{s' | \delta(s, s') > 0\}$. We assume $\mathsf{succ}(s) = \emptyset$ for all non-sink states.

Then, we define a program's probabilistic transition graph as an MC. Note that if a prior distribution of the input values is known, our approach can be easily extended to incorporate the information.

Definition 6 ***(Probabilistic Transition Graph).*** *Given a program* $\mathcal{P} = (I, N, E)$*,* $\mathcal{P}$*'s probabilistic transition graph is an MC* $(\mathcal{S}_t, \delta_t, \mathcal{I}_t)$ *such that*

- $\mathcal{S}_t$ *is* $\mathcal{P}$*'s node set* N*.*
- $\delta_p : \mathcal{S}_t \times \mathcal{S}_t \to \mathbb{R}$ *such that* $\delta_t(s_1, s_2) = 0$ *if* $(s_1, s_2) \notin E$*; otherwise,* $\delta_t(s_1, s_2)$ *is defined as follows. We consider paths up to length* T*, making* $path(s)$ *finite.*

$$\delta_t(s_1, s_2) = \frac{\sum_{p \in path(s_2) \wedge p = \langle ..., s_1, s_2 \rangle} Pr(PC(p))}{\sum_{p \in path(s_1)} Pr(PC(p))} \tag{1}$$

- $\mathcal{I}_t$ *is the initial distribution such that* $\mathcal{I}_t(entry(\mathcal{P})) = 1$*.*

Based on a program's probabilistic transition graph, we can formulate the optimized combination problem with respect to statement coverage by a Markov Decision Process (MDP) with cost. First, we define MDP with cost.

Definition 7 ***(MDP with cost).*** *A Markov Decision Process (MDP) with cost* $\mathcal{M}$ *is a tuple* $(\mathcal{S}, \mathcal{A}, \beta_0, \delta, \mathcal{S}_f, \mathcal{R}, \mathcal{C})$*, where*

- $\mathcal{S}$ *is a set of states.* $\mathcal{S}_f$ *is the set of final states.* $\beta_0 : \mathcal{S} \to \mathbb{R}$ *is the initial distribution function such that* $\Sigma_{s \in \mathcal{S}} \beta_0(s) = 1$*.*
- $\mathcal{A}$ *is a set of actions.*
- $\delta : \mathcal{S} \times \mathcal{A} \to \mathbb{D}$ *is the transition function for a state* s *given an action* a *that gives the distribution of state transitions, i.e.,* $\delta(s, a) : \mathcal{S} \to \mathbb{R}$*, where* $\Sigma_{s_1 \in \mathcal{S}} \delta(s, a)(s_1) = 1$*.*
- $\mathcal{R} : \mathcal{S} \times \mathcal{S} \to \mathbb{R}$ *is the reward function that gives the reward to a state transition.*

- $\mathcal{C} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ *is the cost function that gives the cost of executing an action at a state.*

MDP provides a general formalism for modeling and verifying probabilistic problems. Given an MDP, a policy represents the choices made for the problem and is defined as follows.

Definition 8 ***(Policy).*** *Given an MDP with cost* $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \beta_0, \delta, \mathcal{S}_f, \mathcal{R}, \mathcal{C})$*, a policy* π *is a function in* $\mathcal{S} \to \mathbb{D}$ *such that* π *gives a distribution* $\mathcal{D}_s$ *of the actions in* $\mathcal{A}$ *for a state* s *in* $\mathcal{S}$ *and* $\mathcal{D}_s$ *satisfies* $\sum_{a \in \mathcal{A}} \mathcal{D}_s(a) = 1$*.*

Given an MDP $\mathcal{M}$ and a policy π, the run of the MDP under π is a rollout defined as follows.

Definition 9 ***(Rollout).*** *A rollout* $\zeta \sim \pi$ *for an MDP with cost* $\mathcal{M}$ *is the following sequence of the tuples in* $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \times \mathbb{R} \times \mathbb{R}$

$$\langle (s_0, a_0, s_1, r_0, c_0), ..., (s_{n-1}, a_{n-1}, s_n, r_{n-1}, c_{n-1}) \rangle \tag{2}$$

and the sequence is randomly constructed as follows.

- *Use* β_0 *to sample the initial state* s_0*.*
- *For each* s_i*, use* $\pi(s_i)$ *to sample an action* a_i*. Then, sample the next state* s_{i+1} *by* $\delta(s_i, a_i)$*. Then, the reward* r_i *is the value* $\mathcal{R}(s_i, s_{i+1})$ *and the cost* c_i *is* $\mathcal{C}(s_i, a_i)$*.*
- *Repeat the second step until* s_{i+1} *is in the final state set* $\mathcal{S}_f$*.*

We can calculate the rewards and costs along a rollout. A policy in fact decides a distribution $\mathbb{E}$ of rollouts $\zeta \sim \pi$. Given an MDP with cost, the optimal policy π^* is the one that achieves the maximum reward and minimum cost and is defined as $\pi^* = \arg\max_\pi \mathbb{E}_{\zeta \sim \pi} \left[\sum_{i=0}^{n-1} (r_i - c_i) \right]$.

Then, given an MC of program $\mathcal{P}$, we can define the selective symbolization-based optimized combination with respect to *statement coverage* as an MDP with cost as follows, where $\mathbb{P}(\mathcal{S})$ is the *powerset* of $\mathcal{S}$.

Definition 10 ***(MDP of selective optimized combination).*** *Given a program* $\mathcal{P} = (I, N, E)$ *whose probabilistic transition graph is* $\mathcal{M} = (\mathcal{S}, \delta, \mathcal{I})$*, we define the selective concolic testing with respect to statement coverage as an MDP with cost* $\mathcal{M}_S = (\mathbb{P}(\mathcal{S}), \mathcal{A}_p, \beta_p, \delta_p, \{\mathcal{S}\}, \mathcal{R}_p, \mathcal{C}_p)$*, where*

- $\mathcal{A}_p = \{(R_{I_0}, C_{I_1}) \mid p \in path(\mathcal{P}) \wedge (R_{I_0} \wedge C_{I_1}) = PC(p) \wedge I_0 \cup I_1 = I\}$ is the action set, where each action splits $PC(p)$ into two parts: R_{I_0} (constraints on inputs I_0, handled by random generation) and C_{I_1} (constraints on inputs I_1, solved by constraint solving). We denote by $\mathsf{inputs}(a)$ the program inputs generatable by action a.
  Let $\mathit{\Pi}_a$ be the set of terminated paths reachable by $a = (R_{I_0}, C_{I_1})$, with each

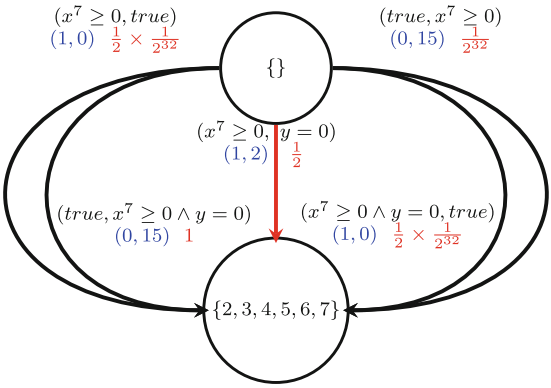


Fig. 2. The *partial* MDP for the selective concolic testing of the example program, where red numbers are probabilities, and blue pairs are the costs.

element $(p, \Pr(p, \mathcal{M}))$ representing a possibly covered terminated path p and its probability. Formally,

$$\Pi_a = \{(p, \Pr(p, \mathcal{M})) \mid \exists i \in \mathsf{inputs}(a) \bullet p = \mathcal{P}(i) \wedge \mathsf{terminated}(p)\},$$

which is derived from $\mathcal{P}$'s probabilistic transition graph.

- β_p is the initial distribution such that $\beta_p(\emptyset) = 1$, *i.e.*, we start from the state where no statement is covered.
- $\delta_p : \mathbb{P}(\mathcal{S}) \times \mathcal{A}_P \to \mathbb{D}$ is the following transition function.

$$\delta_p(N_s, a)(N_t) = \sum_{(p, prob) \in \Pi_a \wedge \mathcal{N}(p) \cup N_s = N_t} prob \tag{3}$$

  N_s is the covered statement set of starting state, and N_t is the covered statement set after the path p terminates.
- Only one final state exists, which is $\mathcal{S}$, *i.e.*, all the **reachable** statements are covered.
- $\mathcal{R}_p : \mathbb{P}(\mathcal{S}) \times \mathbb{P}(\mathcal{S}) \to \mathbb{R}$ is defined as $\mathcal{R}_p(N_1, N_2) = \#(N_2 \setminus N_1)$, *i.e.*, the number of the newly covered statements.
- $\mathcal{C}_p : \mathbb{P}(\mathcal{S}) \times \mathcal{A}_p \to \mathbb{R}$ is defined as $\mathcal{C}_p(s, (R_{I_0}, C_{I_1})) = RC(R_{I_0}) + SC(C_{I_1})$, *i.e.*, the cost of random generation for I_0 and the solving cost of the constraint C_{I_1} in the action.

Note that the action set may be infinite. Our formulation generalizes prior work on combining random testing and symbolic execution [9,44], which restricts actions to those where either I_0 or I_1 is empty.

Assume random generation costs 1, solving $y = 0$ costs 2, and solving $x^7 \geq 0 \wedge y = 0$ (or $x^7 \geq 0$) costs 15. The full MDP $\mathcal{M}_S$ for the program in Fig. 1 has five states and 146 non-zero-probability transitions; Fig. 2 shows only a fragment. For instance, the red transition corresponds to action $(x^7 \geq 0, y = 0)$ with

probability 0.5 and cost 3 (*i.e.*, one random generation plus solving $y = 0$). The transition with action $(true, x^7 \geq 0 \wedge y = 0)$ has probability 1 and cost 15. Other transitions—e.g., $(x^7 < 0, y = 0)$, symmetric to the red one—are omitted for brevity.

Under this formalism, selective concolic testing reduces to computing an optimal policy for a cost-aware MDP [21]. A policy π_s is *successful* if it reaches the final state $\mathcal{S}$ (representing full coverage of reachable statements) with positive probability:

$$\Pr_{\zeta \sim \pi_s} \{s_n = \mathcal{S}\} > 0.$$

Let $\Phi_{\mathcal{S}}$ denote all successful policies for $\mathcal{M}_S$. The optimal policy minimizes expected cost (or equivalently maximizes net reward):

$$\pi^* = \arg\max_{\pi \in \Phi_{\mathcal{S}}} \mathbb{E}_{\zeta \sim \pi} \left[\sum_{i=0}^{n-1} (r_i - c_i) \right].$$

For the MDP in Fig. 2, the optimal policy selects $(x^7 \geq 0, y = 0)$ at the initial state (red transition) and $(x^7 \geq 0, true)$ at state $\{2, 3, 5, 6, 7\}$. Its expected value is 2.5, computed as:

$$0.5 \times (6 - 3) + 0.5 \times (5 + 1 - 3 - 1),$$

corresponding to direct termination and a two-step path via $\{2, 3, 5, 6, 7\}$, respectively.

Discussion. The optimized combination problem reduces to computing an optimal policy in a cost-aware MDP. While solvable in time polynomial in the state-space size [44], this becomes infeasible in practice due to: (1) exponential state-space growth with program size and potentially infinite action spaces; (2) solution-ratio estimation requiring model counting, which is harder than standard constraint solving [18]; (3) infinite paths from loops preventing full transition graph construction; (4) high difficulty in accurately predicting SMT solver costs.

To address these, we propose an approximate algorithm yielding a near-optimized strategy:

- **Local greedy optimization:** Since only partial states/actions are observable during concolic execution, we (i) estimate the probabilistic transition graph from observed traces (Eq. 5, Sect. 4.1) using Laplace smoothing [24] for branch probabilities, and bound input depth to avoid non-termination; and (ii) select branches via local greedy reward (Sect. 4.2).
- **Constraint partitioning:** We split path conditions into two parts, one for SMT solving, one for random generation, via graph partitioning and timeout prediction (Sect. 4.3). Algorithm 3 (Sect. 4.4) uses an ML model [17] to predict solver timeouts and guide partitioning.

Algorithm 1: Selective Concolic Testing

SCT($\mathcal{P}, I_0$)
Input: $\mathcal{P} = (I, N, E)$ is a program, I_0 is the initial input.

```
1:  input ← I_0
2:  M_p ← (S_p, δ_p, I_p)
3:  T ← ∅
4:  while true do
5:      PC, Path_c ← ConcolicExecute(P, input)
6:      T ← T ⊎ OTP(PC)
7:      M_p ← Refine(M_p, Path_c)
8:      PC_n ← ProbSelect(T, M_p)
9:      if PC_n = null then
10:         return
11:     end if
12:     input ← SelectiveSolve(PC_n, I)
13: end while
```

4 Approximate Optimized Algorithm

Algorithm 1 shows the high-level approach of selective concolic testing. The inputs are the program $\mathcal{P} = (I, N, E)$ under analysis and the initial input I_0.

We use a map $\mathcal{T}$ to store all the off-the-path branches and their path conditions. In the beginning, $\mathcal{T}$ is empty (Line 3). Then, the algorithm enters the main loop for path exploration. Line 5 executes $\mathcal{P}$ under the current input in a concolic manner [22,38]. The execution returns the current path condition PC and the path $Path_c$ of the current execution. After the execution terminates, the algorithm adds all the off-the-path branches of PC to $\mathcal{T}$ (Line 6). $OTP(PC)$ represents the set of off-the-path branch mappings that are defined as $\{\mathcal{B}(\neg C_j) \mapsto (\bigwedge_{k\in\{1,\ldots,j-1\}} C_k) \wedge \neg C_j \mid 1 \leq j \leq n\}$, where PC is $\bigwedge_{i\in\{1,\ldots,n\}} C_i$ and $\mathcal{B}(\neg C_j)$ denotes the branch (with context information) of the condition $\neg C_j$.

Based on $\mathcal{P}$'s approximate probabilistic transition graph (Sect. 4.1), the algorithm then selects (Sect. 4.2) an off-the-path branch b from $\mathcal{T}$ (Line 8) and returns b's path condition PC_n, *i.e.*, $\mathcal{T}[b]$. In case there exists no unexplored branch, ProbSelect returns *null*, *i.e.*, the path exploration is done, and the algorithm terminates (Line 9–11). Otherwise, the algorithm solves the path condition PC_n by the selective solving algorithm (Line 12) to generate the program input for the next concolic execution. The main loop continues until the computation resource exhausts, *e.g.*, reaching the time limit, which is omitted due to the space limit.

4.1 Approximating the Program's MC

In Algorithm 1, $\mathcal{M}_p$ is the initial probabilistic transition graph of $\mathcal{P}$ at the beginning, where $\mathcal{S}_p$ is N, $\mathcal{I}_p(I) = 1$, and δ_p is as follows.

$$\delta_p(s_1, s_2) = \begin{cases} 0 & (s_1, s_2) \notin E \\ \frac{1}{\#\mathsf{succ}(s_1)} & otherwise \end{cases} \tag{4}$$

Then, after each run of the program, we use the current path $Path_c$ to refine $\mathcal{M}_p$'s transition probabilities (Line 7). δ_p is refined by Laplace estimation [24] and defined as follows, where $\mathsf{EC}(s_1, s_2)$ and $\mathsf{EC}(s_1)$ represents how many times the transition (s_1, s_2) and the statement s_1 have been visited by the executions during concolic testing, respectively.

$$\delta_p(s_1, s_2) = \begin{cases} 0 & (s_1, s_2) \notin E \\ \frac{\mathsf{EC}(s_1,s_2)+1}{\mathsf{EC}(s_1)+\#succ(s_1)} & otherwise \end{cases} \tag{5}$$

4.2 Probability Based Branch Selection

Our algorithm uses transition probabilities to select the off-the-path branch to explore. The key idea is to use the transition probabilities to calculate each branch's reward, which reflects the branch's ability to cover new statements. Here, inspired by the MDP definition of optimized combination (Definition 10) and the value iteration based method [21] for calculating the optimal policy of MDP, we calculate the reward of each statement s (or node) in $\mathcal{M}_p$ as follows.

$$\mathcal{R}(s, \mathcal{M}_p) = \begin{cases} \Sigma_{t \in \mathsf{succ}(s)}(\delta_p(s,t) \times \mathcal{R}(t)) & s\ is\ covered \\ 1 + \Sigma_{t \in \mathsf{succ}(s)}(\delta_p(s,t) \times \mathcal{R}(t)) & otherwise \end{cases} \tag{6}$$

Hence, the rewards of uncovered statements are larger than those of covered statements. Then, ProbSelect returns the path condition of the off-the-path branch in $\mathcal{T}$, which is defined as $\arg\max_{(b,PC) \in \mathcal{T}} \mathcal{R}(\mathsf{FST}(b), \mathcal{M}_p)$, where $\mathsf{FST}(b)$ represents the *first statement* of the branch b's basic block.

4.3 Selective Solving Algorithm

We want to optimally combine SMT solving and random generation at the constraint solving level. We formalize the problem as follows. Given a path constraint PC of a program's input I, we need to find the optimized partition of I that is defined as follows, where $\mathcal{P}_I = \{(R, S) \mid R \cup S = I\}$ and $PC \downarrow R$ denotes PC's projection on R.

$$\arg \max_{(R,S) \in \mathcal{P}_I} \frac{Pr(PC \downarrow R)}{SC(PC \downarrow S)} \tag{7}$$

The optimized partition maximizes the probability of random testing and minimizes the solving cost of constraint solving.

Algorithm 2 shows the main procedure of selective solving. The inputs are the path condition PC and the set I of input variables. The output is the solving result, which has the following three cases: `SAT` with a solution, `UNSAT`, and `UNKNOWN`. The algorithm first separates the path condition PC into two parts

Algorithm 2: Selective Solving Algorithm

SelectiveSolve(PC, I)
Input: $\mathcal{PC}$ is a path condition, I is the set of the input variables in PC.
1: $(PC_r, PC_c) \leftarrow$ Partition(PC, I)
2: $(r_1, S_c) \leftarrow$ SMTSolving(PC_c)
3: **if** $r_1 =$ `UNSAT` $\vee\, r_1 =$ `UNKNOWN` **then**
4: **return** r_1
5: **end if**
6: $(r_2, S_r) \leftarrow$ RandomGeneration(PC, PC_r, S_c)
7: **if** $r_2 =$ `UNKNOWN` **then**
8: **return** r_2
9: **end if**
10: $S \leftarrow S_c \uplus S_r$
11: **return** S

PC_r and PC_c (Line 1). Algorithm 3 handles the separation. PC_r will be solved by random generation (Line 6), and the algorithm uses the underlying SMT solver to solve PC_c (Line 2). If the SMT solver produces `UNSAT` or `UNKNOWN` (*e.g.*, timeout), the algorithm returns the result directly (Line 4) because the constraint PC_c's `UNSAT` implies the PC's unsatisfiability. If both PC_c and PC_r are `SAT` and can be solved, the algorithm merges their solutions S_c and S_r. $S_c \uplus S_r$ can be defined as follows, where S_c and S_r are the solution maps that give a value to a variable, and $v \hat{\in} S$ represents that v is a key in S.

$$S_c \uplus S_r[v] := \begin{cases} S_r[v] & v \hat{\in} S_r \\ S_c[v] & v \hat{\in} S_c \\ \texttt{Undefined}\ otherwise \end{cases} \tag{8}$$

In the algorithm, some input variables may appear in both PC_r and PC_c after partition, which is why we pass the solution of PC_c when using random generation for solving PC_r (Line 6). During the random generation, we will use the solution S_c to assign the values of the PC_r's variables that also appear in PC_c. In our implementation, we employ fuzzing-based solving [33] to solve PC_r. Our selective solving algorithm is sound but incomplete. The input path condition PC may be satisfiable, but our algorithm may produce `UNKNOWN` because we employ random generation (or fuzzing in our implementation) to solve PC_r.

4.4 Partition Algorithm

Algorithm 3 partitions a path condition PC over input variables I into two parts: PC_r (for random generation) and PC_c (for SMT solving), such that $PC = PC_r \wedge PC_c$. The core idea is to (1) model variable dependencies in PC as a weighted graph, (2) partition it using a graph clustering algorithm [29], and (3) assign each sub-constraint to either PC_r or PC_c based on an offline-trained ML model that predicts SMT solver timeout.

Algorithm 3: Constraint Partition

Partition(PC, I)
Input: PC is a path condition set, and I is the set of the variables in PC.
1: $\mathcal{E} \leftarrow \{(v_1, v_2) \mapsto 0 \mid v_1 \neq v_2 \wedge \{v_1, v_2\} \subseteq I\}$
2: $\mathcal{G} \leftarrow (I, \mathcal{E})$
3: **for** condition **do**
4: $V \leftarrow \mathit{Variables}(C_i)$
5: **for** each $(v_1, v_2) \in V \times V$ **do**
6: **if** $v_1 \neq v_2$ **then**
7: $\mathcal{E}[(v_1, v_2)] \leftarrow \mathcal{E}[(v_1, v_2)] + 1$
8: **end if**
9: **end for**
10: **end for**
11: $GS \leftarrow$ GraphPartition$(\mathcal{G})$
12: $PC_r, PC_c \leftarrow true, true$
13: **for** each $g \in GS$ **do**
14: $PC_g \leftarrow PC \downarrow g$
15: **if** Predicte(PC_g) is timeout **then**
16: $PC_r \leftarrow PC_r \wedge PC_g$
17: **else**
18: $PC_c \leftarrow PC_c \wedge PC_g$
19: **end if**
20: **end for**
21: **return** (PC_r, PC_c)

We construct a relation graph $\mathcal{G} = (I, \mathcal{E})$, where each edge (v_1, v_2) is weighted by the number of atomic predicates in PC containing both v_1 and v_2:

$$w(v_1, v_2) = \#\{C_i \mid v_1 \in C_i \wedge v_2 \in C_i\}. \tag{9}$$

Lines 1–10 initialize $\mathcal{E}$ and populate edge weights by iterating over all atomic conditions $C_i \in PC$ and incrementing weights for every distinct variable pair in $\mathit{Variables}(C_i)$. The graph $\mathcal{G}$ is then partitioned into subgraphs GS via GraphPartition (Line 11). For each subgraph $g = (V_g, E_g) \in GS$, we project PC onto g to obtain $PC_g = \bigwedge\{C_i \in PC \mid \mathit{Variables}(C_i) \subseteq V_g\}$ (Line 14). In Lines 13–20, an SVM model [16], trained offline on BVFP formulas, predicts whether solving PC_g will time out. The feature vector is built via Bag-of-Words over SMT operators (e.g., counts of `FAdd`, `FMul`), including the number of BVFP operations. If timeout is predicted, PC_g is added to PC_r; otherwise, to PC_c.

Example: Consider the path condition $x^7 > y \wedge y = z \wedge y = w + 2 \wedge z = w + w$ over input variables $\{x, y, z, w\}$.

We construct a relation graph $\mathcal{G}$ where vertices are variables and edge weights reflect co-occurrence in atomic predicates. Applying graph partitioning yields two subgraphs: $g_1 = \{x, y\}$ and $g_2 = \{y, z, w\}$. The projection of the path condition onto g_1 is $PC_{g_1} = x^7 > y$, which our predictor classifies as likely to

Table 1. Benchmark Introduction

Benchmark	LoC	#Fun	Info
GNU GSL [3]	70725	421	A scientific computing library
Cephes [1]	18307	114	A mathematical routines library
FDLIBM [2]	18499	41	A portable C math library
Total	107531	576 functions	

time out during SMT solving. The projection onto g_2 is $PC_{g_2} = y = z \wedge y = w + 2 \wedge z = w + w$, predicted to be efficiently solvable.

Thus, the path condition is partitioned as:

$$\underbrace{x^7 > y}_{PC_r} \wedge \underbrace{y = z \wedge y = w + 2 \wedge z = w + w}_{PC_c} \tag{10}$$

Solving PC_c with an SMT solver yields a concrete assignment, e.g., $\{y \mapsto 4, z \mapsto 4, w \mapsto 2\}$ (corresponding to S_c in Algorithm 2). To satisfy PC_r, we fix $y = 4$ and use random generation to find x such that $x^7 > 4$, which is straightforward due to the high solution density of this inequality.

5 Evaluation

We implemented our approximate algorithm for C programs in a concolic execution framework [38] built on KLEE [10]. Complex constraints are handled by integrating JFS [33] as the random-testing engine. Path conditions (in KQuery [4]) are partitioned via METIS [29] using their relational graph. To predict solver timeouts, we extract syntactic features with a Bag-of-Words model [48] (counting operator occurrences) and train an SVM using LIBSVM [16]. The model was trained on the SMTLIB2 QF_BVFP benchmark [6] that does not intersect with the experimental evaluation, with an accuracy of 91.1%, avoiding training leakage. We use Z3 [37] as the main solver for its QF_ABVFP support and extended JFS to cover more SMTLIB2 operations.

- **RQ1 (Effectiveness):** How does our method compare to existing optimized combination and heuristic approaches in statement coverage?
- **RQ2 (Efficiency):** How much faster does our method achieve the same statement coverage as prior work?

5.1 Experimental Setup

Benchmarks. Table 1 shows two types of benchmarks for evaluation. **GSL & Cephes** contain complex non-linear floating-point path conditions and intensive control flow, making them challenging for symbolic execution. We evaluate 421

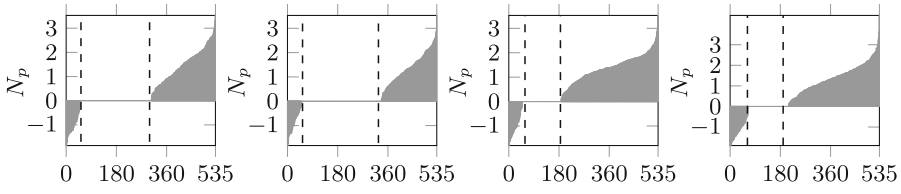


Fig. 3. Experimental results of #**Cov** on GSL and Cephes benchmark

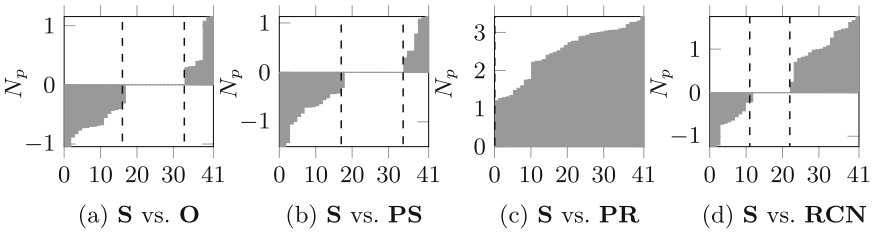


Fig. 4. Experimental results of #**Cov** on FDLIBM benchmark

GSL and 114 Cephes functions (total 535), including both hard cases and small functions that allow full path exploration. **FDLIBM** features linear conditions with low-level bit manipulations typical in embedded systems. To avoid redundancy from type-duplicated logic (e.g., `cos(float)` vs `cos(double)`), we select 41 unique functions.

Baselines. We compare our method (**S**) against four baselines:

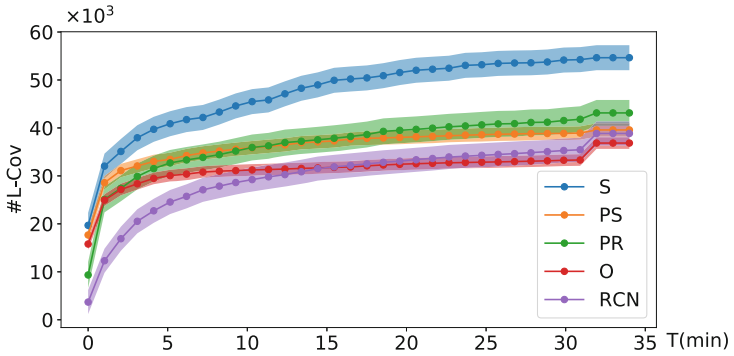
- **O**: optimized combination without selective symbolization [44];
- **PS**: probabilistic branch selection with pure Z3 solver;
- **PR**: probabilistic branch selection with pure JFS solver;
- **RCN**: KLEE's default random-cover-new strategy with Z3 solver.

Our evaluation addresses three goals: (1) **Overall performance**: Compare coverage and efficiency of **S** vs. baselines; (2) **Branch selection impact**: Evaluate probabilistic selection via **PS** vs. **RCN**; (3) **Selective solving benefit**: Assess our hybrid solver by comparing **S**, **PS**, and **PR**.

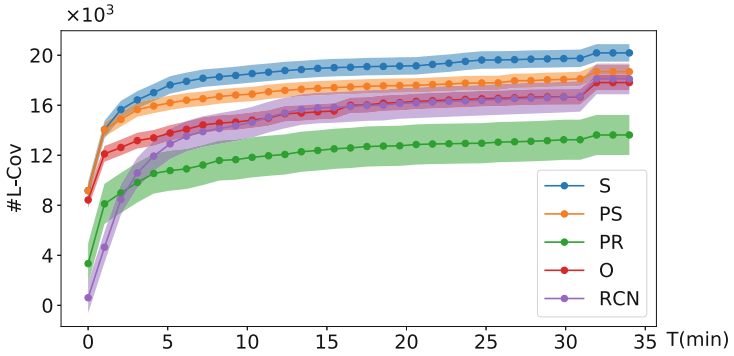
All experiments run for 30 min, repeated 5 times on a 104-core server (Intel Xeon Platinum 8269CY @2.50GHz, 194GB RAM, Ubuntu 18.04).

5.2 Experimental Results

GSL and Cephes. Figure 3 shows statement coverage results across 535 functions, with programs sorted by increasing coverage and relative improvement N_p



(a) GSL #**Cov** trend



(b) Cephes #**Cov** trend

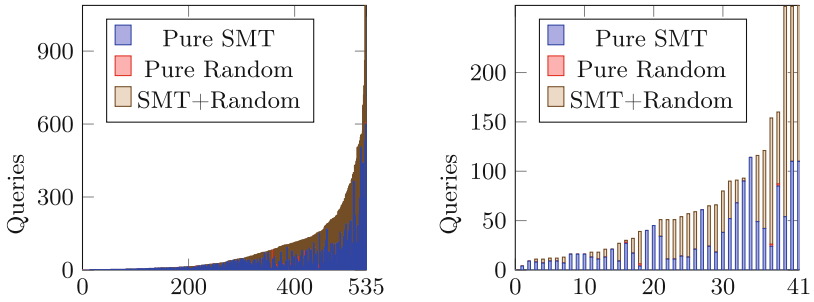
Fig. 5. Trends of covered statements on **GSL** and **Cephes**. Mean values and 95% confidence intervals over 5 runs are shown in solid lines and shadows, respectively.

(log scale) defined as:

$$N_p = \begin{cases} \log_{10}(R_p \times 100 + 1) & R_p \geq 0 \\ -\log_{10}(-R_p \times 100 + 1) & R_p < 0 \end{cases} \tag{11}$$

where $R_p = (L_p^{\mathbf{S}} - L_p^b)/L_p^b$. $N_p = 0$ indicates equal coverage; $N_p = 1$ and 2 correspond to 10% and 100% gains.

Our method **S** outperforms all baselines on most functions. Compared to **O**, **PS**, **PR**, and **RCN**, **S** achieves higher coverage in 230, 199, 348, and 326 functions, with average improvements of 36.89%, 28.43%, 31.85%, and 31.46%, respectively. Underperformance on some functions stems from inaccurate timeout predictions, causing unnecessary random solving; small functions often terminate early.



(a) Partition of **GSL and Cephes** (b) Partition of **FDLIBM**

Fig. 6. Partition information of the benchmarks

FDLIBM. Figure 4 shows results on FDLIBM's linear but complex constraints. **S** performs comparably to **PS** (−3.44%) and significantly better than **PR** (157.42%), as random solving struggles with equalities. It matches **O** (−0.05%) and **RCN** (3.44%), both of which favor SMT solving. Despite timeout prediction errors limiting gains, **S** remains competitive, demonstrating robustness across constraint types.

> **Answer to RQ1: S** significantly improves coverage on GSL/Cephes and matches SMT-based methods on FDLIBM, confirming its effectiveness on real-world floating-point programs.

Efficiency. Figure 5 shows cumulative covered statements over time (mean ±95% CI). On GSL, **S** achieves speedups of 12.08×, 10.88×, 3.88×, and 8.76× over **O**, **PS**, **PR**, and **RCN** at peak coverage; similar gains are seen on Cephes. Baselines exhibit delayed coverage spikes after 30 min due to poor state prioritization, whereas **S**, **PS**, and **PR** show smoother, earlier growth thanks to probabilistic branch selection.

> **Answer to RQ2: S** is consistently more efficient at achieving the same coverage.

Constraint Partitioning. Figure 6 shows solving mode distribution. On GSL/-Cephes, 24.03% of constraints used hybrid solving (SMT+random), versus 75.55% pure SMT and 0.42% pure random. On FDLIBM, hybrid use was 50.82%, reflecting its linear nature. Since constraint solving dominates (>90% of runtime), selective symbolization is essential. The low pure-random usage on FDLIBM confirms SMT's suitability, yet prediction errors caused unnecessary hybrid attempts, explaining **S**'s slight underperformance vs. **PS**.

Our method excels on programs with mixed constraint complexity (e.g., GSL and Cephes's hypergeometric functions), where partitioning separates hard

floating-point constraints from simple branches. When all constraints are uniformly complex (e.g., FDLIBM's linear cases), gains diminish as random testing becomes less effective.

Limitations. Our approach assumes: (1) program variables have finite domains; (2) the control-flow graph contains no unreachable code. While these hold for mathematical libraries, real-world programs may require preprocessing to satisfy these conditions.

6 Related Work

Our work relates to hybrid approaches combining symbolic execution with fuzzing or random testing [36,40,44,51]. The term **Selective Concolic Testing** was previously used by Lin *et al.* [34] in the context of hardware security for detecting Trojans in SystemC designs. While both approaches aim to improve concolic testing efficiency through selection, they differ fundamentally in objective and mechanism. Lin *et al.* selectively restrict test generation to user-specified code regions to avoid redundant path exploration. In contrast, our work provides a theoretical framework for selective input symbolization, and the practical algorithm partitions based on a cost model derived from machine learning and Markov decision processes.

Concolic testing [22,38] inherently blends both by using concrete values to simplify constraints. Majumdar and Sen [36] interleave random and concolic testing, switching to the latter when coverage stalls; Driller [40] applies a similar idea to binary analysis. Wang *et al.* [44] first formalize the optimized combination for statement coverage as a Markov Decision Process (MDP) [21], while DigFuzz [51] uses probabilistic path prioritization. In contrast, our key insight is that which inputs to symbolize critically affects efficiency. To our knowledge, we are the first to formalize the optimized combination problem under selective symbolization. Unlike Cha *et al.* [15], who mine templates for selective symbolization online, we formally define the optimal policy and use fuzzing for non-symbolized inputs. Marco [25] also models concolic testing as an MDP but uses path divergence probability as cost; we instead model constraint-solving difficulty, making our approach more effective for programs with complex (e.g., floating-point) constraints. Our method complements existing symbolic execution techniques. Search heuristics [7,35,45,50] prioritize branches for coverage or bug finding, while path pruning methods [19,27,42,46] eliminate irrelevant paths, both orthogonal to our constraint-level optimization. We also build on constraint-solving optimizations, such as formula simplification and caching [10], solution reuse [28,43], and domain-specific solvers [17,39,49]. Our algorithm uniquely integrates SMT solving with fuzzing at the constraint level, aligning with hybrid fuzzing [26,30,32,47].

7 Conclusion

Optimally combining symbolic execution and random testing is challenging. We formalize selective concolic testing as an MDP policy optimization problem and propose an approximate algorithm that partitions path constraints: one part is solved by an SMT solver, the other via random generation. Implemented atop KLEE and JFS, our prototype demonstrates significant gains in both effectiveness and efficiency.

Acknowledgments. This research was supported by National Key R&D Program of China (No. 2022YFB4501903) and the NSFC Program (No. 62172429, 62032024, and 62372162), and Jun Sun was supported by the Ministry of Education, Singapore under its Academic Research Fund Tier 2 (Award ID: T2EP20222-0037). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore.

Data Availability Statement. Our artifact is available at the following URL: https://doi.org/10.5281/zenodo.18603532.

References

1. Cephes mathematical function library (2025). https://www.netlib.org/cephes/. Accessed 12 Nov 2025
2. Freely distributable libm (2025). http://www.netlib.org/fdlibm/. Accessed 12 Nov 2025
3. Gsl - gnu scientific library (2025). https://www.gnu.org/software/gsl/. Accessed 12 Nov 2025
4. Klee's kquery language (2025). https://klee.github.io/docs/kquery/. Accessed 12 Nov 2025
5. Qf_lia logic (2025). https://smtlib.cs.uiowa.edu/logics-all.shtml#QF_LIA. Accessed 12 Nov 2025
6. Smt-lib benchmark (2025). https://smtlib.cs.uiowa.edu/benchmarks.shtml. Accessed 12 Nov 2025
7. Avgerinos, T., Cha, S.K., Hao, B.L.T., Brumley, D.: AEG: automatic exploit generation. In: NDSS 2011, San Diego, California, USA, 6th February–9th February 2011 (2011)
8. Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Comput. Surv. **51**(3), 50:1–50:39 (2018)
9. Burnim, J., Sen, K.: Heuristics for scalable dynamic test generation. In: 23rd IEEE/ACM ASE 2008), L'Aquila, Italy, 15–19 September 2008, pp. 443–446 (2008)
10. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: 8th USENIX OSDI 2008, San Diego, California, USA, 8–10 December 2008 Proceedings, pp. 209–224 (2008)
11. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. In: Proceedings of the 13th ACM CCS 2006, Alexandria, VA, USA, 30 October–3 November 2006, pp. 322–335 (2006)

12. Cadar, C., et al.: Symbolic execution for software testing in practice: preliminary assessment. In: Proceedings of the 33rd ICSE 2011, Waikiki, Honolulu, HI, USA, 21–28 May 2011, pp. 1066–1071 (2011)
13. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM **56**(2), 82–90 (2013)
14. Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing mayhem on binary code. In: IEEE SP 2012, San Francisco, California, USA, 21–23 May 2012, pp. 380–394 (2012)
15. Cha, S., Lee, S., Oh, H.: Template-guided concolic testing via online learning. In: Huchard, M., Kästner, C., Fraser, G. (eds.) Proceedings of the 33rd ASE 2018, Montpellier, France, 3–7 September 2018, pp. 408–418. ACM (2018)
16. Chang, C.C., Lin, C.J.: LIBSVM: a library for support vector machines. ACM Trans. Intell. Syst. Technol. **2**, 27:1–27:27 (2011), software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm
17. Chen, Z., et al.: Synthesize solving strategy for symbolic execution. In: ISSTA 2021, Virtual Event, Denmark, 11–17 July 2021, pp. 348–360 (2021)
18. Chistikov, D., Dimitrova, R., Majumdar, R.: Approximate counting in SMT and value estimation for probabilistic programs. In: TACAS 2015, London, UK, 11–18 April 2015. Proceedings, pp. 320–334 (2015)
19. Cui, H., Hu, G., Wu, J., Yang, J.: Verifying systems rules using rule-directed symbolic execution. In: ASPLOS 2013, Houston, TX, USA, 16–20 March 2013, pp. 329–342 (2013)
20. Deng, X., Lee, J., Robby: Bogor/Kiasan: a k-bounded symbolic execution for checking strong heap properties of open systems. In: 21st IEEE/ACM ASE 2006, Tokyo, Japan, 18–22 September 2006, pp. 157–166 (2006)
21. Feinberg, E., Shwartz, A. (eds.): Handbook of Markov Decision Processes - Methods and Applications. Kluwer International Series (2002)
22. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Proceedings of the ACM SIGPLAN PLDI 2005, Chicago, IL, USA, 12–15 June 2005, pp. 213–223 (2005)
23. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: NDSS 2008, San Diego, California, USA, 10th February–13th February 2008 (2008)
24. Hartley, H.O.: Contributions to Survey Sampling and Applied Statistics. Academic Press, London (1978)
25. Hu, J., Duan, Y., Yin, H.: Marco: a stochastic asynchronous concolic explorer. In: Proceedings of the 46th IEEE/ACM ICSE 2024, Lisbon, Portugal, 14–20 April 2024, pp. 59:1–59:12. ACM (2024)
26. Huang, H., Yao, P., Wu, R., Shi, Q., Zhang, C.: Pangolin: incremental hybrid fuzzing with polyhedral path abstraction. In: 2020 IEEE SP 2020, San Francisco, CA, USA, 18–21 May 2020, pp. 1613–1627 (2020)
27. Jaffar, J., Murali, V., Navas, J.A.: Boosting concolic testing via interpolation. In: The ACM SIGSOFT ESEC/FSE 2013, Saint Petersburg, Russian Federation, 18–26 August 2013, pp. 48–58 (2013)
28. Jia, X., Ghezzi, C., Ying, S.: Enhancing reuse of constraint solutions to improve symbolic execution. In: ISSTA 2015, Baltimore, MD, USA, 12–17 July 2015, pp. 177–187 (2015)
29. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. **20**(1), 359–392 (1998)
30. Kim, K., Jeong, D.R., Kim, C.H., Jang, Y., Shin, I., Lee, B.: HFL: hybrid fuzzing on the Linux kernel. In: 27th NDSS 2020, San Diego, California, USA, 23–26 February 2020 (2020)

31. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
32. Le, H.M.: LLVM-based hybrid fuzzing with libkluzzer (competition contribution). In: FASE 2020, Dublin, Ireland, 25–30 April 2020, Proceedings, pp. 535–539 (2020)
33. Liew, D., Cadar, C., Donaldson, A.F., Stinnett, J.R.: Just fuzz it: solving floating-point constraints using coverage-guided fuzzing. In: Proceedings of the ACM ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, 26–30 August 2019, pp. 521–532 (2019)
34. Lin, B., Chen, J., Xie, F.: Selective concolic testing for hardware trojan detection in behavioral systemc designs. In: 2020 Design, Automation & Test in Europe Conference & Exhibition, DATE 2020, Grenoble, France, 9–13 March 2020, pp. 19–24. IEEE (2020)
35. Ma, K.-K., Yit Phang, K., Foster, J.S., Hicks, M.: Directed symbolic execution. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 95–111. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23702-7_11
36. Majumdar, R., Sen, K.: Hybrid concolic testing. In: 29th ICSE 2007, Minneapolis, MN, USA, 20–26 May 2007, pp. 416–426 (2007)
37. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
38. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: The 13th ACM ESEC/SIGSOFT FSE 2005, Lisbon, Portugal, 5–9 September 2005, pp. 263–272 (2005)
39. Shuai, Z., Chen, Z., Zhang, Y., Sun, J., Wang, J.: Type and interval aware array constraint solving for symbolic execution. In: 30th ACM SIGSOFT ISSTA 2021, Virtual Event, Denmark, 11–17 July 2021, pp. 361–373 (2021)
40. Stephens, N., et al.: Driller: augmenting fuzzing through selective symbolic execution. In: NDSS 2016, San Diego, California, USA, 21–24 February 2016 (2016)
41. Tillmann, N., de Halleux, J.: Pex-white box test generation for .net. In: Tests and Proofs - 2nd International Conference, TAP 2008, Prato, Italy, 9–11 April 2008. Proceedings, pp. 134–153 (2008)
42. Trabish, D., Mattavelli, A., Rinetzky, N., Cadar, C.: Chopped symbolic execution. In: Proceedings of the 40th ICSE 2018, Gothenburg, Sweden, 27 May–03 June 2018, pp. 350–360 (2018)
43. Visser, W., Geldenhuys, J., Dwyer, M.B.: Green: reducing, reusing and recycling constraints in program analysis. In: 20th ACM SIGSOFT SIGSOFT/FSE 2012, Cary, NC, USA, 11–16 November 2012, p. 58 (2012)
44. Wang, X., Sun, J., Chen, Z., Zhang, P., Wang, J., Lin, Y.: Towards optimal concolic testing. In: ICSE 2018, Gothenburg, Sweden, 27 May–03 June 2018, pp. 291–302 (2018)
45. Xie, T., Tillmann, N., de Halleux, J., Schulte, W.: Fitness-guided path exploration in dynamic symbolic execution. In: IEEE/IFIP DSN 2009, Estoril, Lisbon, Portugal, 29 June–2 July 2009, pp. 359–368 (2009)
46. Yu, H., Chen, Z., Wang, J., Su, Z., Dong, W.: Symbolic verification of regular properties. In: Proceedings of the 40th ICSE 2018, Gothenburg, Sweden, 27 May–03 June 2018, pp. 871–881 (2018)
47. Yun, I., Lee, S., Xu, M., Jang, Y., Kim, T.: QSYM: a practical concolic execution engine tailored for hybrid fuzzing. In: 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, 15–17 August 2018, pp. 745–761 (2018)
48. Zhang, Y., Jin, R., Zhou, Z.: Understanding bag-of-words model: a statistical framework. Int. J. Mach. Learn. Cybern. **1**(1–4), 43–52 (2010)

49. Zhang, Y., Chen, Z., Shuai, Z., Zhang, T., Li, K., Wang, J.: Multiplex symbolic execution: exploring multiple paths by solving once. In: 35th IEEE/ACM ASE 2020, Melbourne, Australia, 21–25 September 2020, pp. 846–857 (2020)
50. Zhang, Y., Chen, Z., Wang, J., Dong, W., Liu, Z.: Regular property guided dynamic symbolic execution. In: ICSE 2015, Florence, Italy, 16–24 May 2015, vol. 1, pp. 643–653 (2015)
51. Zhao, L., Duan, Y., Yin, H., Xuan, J.: Send hardest problems my way: probabilistic path prioritization for hybrid fuzzing. In: NDSS 2019, San Diego, California, USA, 24–27 February 2019 (2019)