MARCO: A Stochastic Asynchronous Concolic Explorer

Jie Hu
University of California, Riverside
jhu066@ucr.edu

Yue Duan
Singapore Management University
yueduan@smu.edu.sg

Heng Yin
University of California, Riverside
heng@cs.ucr.edu

ABSTRACT

Concolic execution is a powerful program analysis technique for code path exploration. Despite recent advances that greatly improved the efficiency of concolic execution engines, path constraint solving remains a major bottleneck of concolic testing. An intelligent scheduler for inputs/branches becomes even more crucial. Our studies show that the previously under-studied branch-flipping policy adopted by state-of-the-art concolic execution engines has several limitations. We propose to assess each branch by its potential for new code coverage from a global view, concerning the path divergence probability at each branch. To validate this idea, we implemented a prototype MARCO and evaluated it against the state-of-the-art concolic executor on 30 real-world programs from Google's Fuzzbench, Binutils, and UniBench. The result shows that MARCO can outperform the baseline approach and make continuous progress after the baseline approach terminates.

ACM Reference Format:
Jie Hu, Yue Duan, and Heng Yin. 2024. MARCO: A Stochastic Asynchronous Concolic Explorer. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3623301

1 INTRODUCTION

Concolic execution (CE), which conducts concrete and symbolic execution of the program under test (PUT) simultaneously, is a program testing technique used for code exploration and vulnerability detection. Unlike dynamic symbolic execution (DSE), which explores the program space by using symbolic inputs [7], concolic execution is performed with a concrete input exercising a concolic path consisting of branches that are dependent on a subset of input bytes. Each input-dependent branch in the concolic path has two directions: 1) a visited direction that is traversed by the concrete execution; 2) an unvisited direction that can potentially lead to a new path. Concolic execution effectively explores the program space by generating new inputs that traverse these unvisited directions of input-dependent branches.

Although powerful, concolic execution is known to be costly. As a result, many efforts have been made to improve the runtime efficiency of CE over the past few years, in terms of both constraint collection and constraint solving [11, 12, 35, 36, 50]. In contrast, another essential component in concolic execution, branch-flipping policy, has not yet received enough attention. A branch-flipping policy dictates which symbolic branch needs to be flipped to generate a new testcase traversing the flipped branch. State-of-the-art (SOTA) CE engines employ a very restrictive branch-flipping policy – to flip only a very small fraction of symbolic branches that are most likely to reach new code coverage – in order to suppress testcase/path explosion problems, where the number of generated testcases quickly surpasses CE's processing capacity.

In this paper, we conduct the first study on this branch-flipping policy and have a few unique observations. On the one hand, we show that this policy is too strict, misses many good branches that could lead to much higher code coverage. On the other hand, this policy is not as effective as expected since only a small fraction (on average 27%) of branches selected by this policy can actually lead to new code coverage (see §2.3). Consequently, we observe that this rigid and nearsighted branch-flipping policy significantly undermines the effectiveness of CE.

Moreover, we show that the path divergence problem [2] (i.e., the testcase generated by CE does not follow the expected path) can be as high as 50% in practice and is a norm rather than an exception due to the imperfect design and implementation of CE. Therefore, we argue that a good branch-flipping policy needs to model the path divergence on each symbolic branch when selecting the next branch to flip.

To overcome the limitations, we propose a global-view new-coverage directed branch scheduling algorithm for concolic execution. To find out which symbolic branch is the best to flip, we estimate the potential of each symbolic branch (i.e., how likely we can reach new code coverage by flipping this branch) and select the branch with the highest potential. Specifically, we model the concolic execution as a Markov process: each branch transition is a probabilistic event, and an execution path is a sequence of branch transitions, and thus a sequence of probabilistic events. To obtain a global view of all testcases, we observe the executions of all testcases and construct a *stochastic* Concolic State Transition Graph (CSTG) to characterize transition probabilities between states and estimate the probability of a given branch reaching any unvisited states. We refer to this probability as *reachability score*. This reachability score is further dampened by the path divergence rate observed on this branch.

To select the best branch (i.e., one with the highest reachability score) to flip, our branch selection must be *asynchronous*. When encountering a symbolic branch, the existing CE engines decide synchronously whether to flip it based on the historical information collected up to this point. This decision, however, might not be globally optimal because a seemingly good branch to flip might have already been traversed by the remaining execution of the current testcase or the remaining testcases that have not been processed yet. Therefore, we propose to process all testcases to maintain an up-to-date global view (in the form of CSTG) and then asynchronously select the best branch to flip. To do so, we develop

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3623301

an efficient concolic state saving and restoring mechanism. We save the symbolic expression table and branch dependency information for quick reloading after the highest potential state is identified.

To evaluate the efficacy of this idea, we implement a prototype called MARCO[1], atop SymSan [11]. We evaluated MARCO on 16 real-world programs and 71 programs from the DARPA Cyber Grand Challenge (CGC) binary set to demonstrate that MARCO, on average, increases edge coverage by 13.03%. For 3 out of the 11 programs where MARCO finds more coverage, it also covers all edge coverage found by SymSan. We further evaluate its bug detection efficiency on 14 programs from Unifuzz [29]. The result shows that our approach can find 33.52% more unique bugs than the SOTA CE engine SymSan. Furthermore, MARCO can uniquely find more than twice of bugs than SymSan does. On 5 of the tested programs, MARCO finds more unique bugs in 12h than any of the seven fuzzers evaluated in UniFuzz (excluding QSYM, which is configured as a hybrid fuzzer) can find in 24h experimental runs.

The contributions of this paper are summarized as follows:

- We evaluate the state-of-the-art branch-flipping policy and reveal several important yet unreported limitations.
- We propose a stochastic and asynchronous branch scheduling algorithm that is able to effectively pick the most promising branch for new input generation.
- We implement a prototype MARCO and evaluate its efficacy on 16 real-world applications. The experimental results demonstrate that MARCO can constantly outperform the SOTA in terms of coverage finding and bug detection.
- We open-source the implementation of our prototype at https://zenodo.org/record/8339481.

2 BACKGROUND AND MOTIVATIONS

In this section, we provide the background knowledge about symbolic/concolic execution and existing branch-flipping policies and further motivate our work by stating four key observations.

2.1 Symbolic Execution

Symbolic Execution (SE) is an automated program testing technique that aims to maximize code coverage by generating specific inputs to satisfy every condition check that is dependent on the input within the program under test. With SE, the program is executed with symbolic expressions instead of concrete values. An SE engine maintains 1) the mapping between program variables and symbolic expressions, and 2) a set of path predicates imposed by the sequence of branches visited along the execution path.

Two types of SE are extensively researched: 1) online symbolic execution and 2) concolic execution. Online symbolic execution engines, such as KLEE [7] and S2E [15], explore the program space via state forking: when encountering a branch point (whose direction is dependent on the input), an SE engine will fork a new state to explore the opposite branch direction (if it is feasible). As a result, the number of states grows exponentially, leading to the state explosion problem. To tackle the state explosion problem, some recent works [25, 31] resorted to machine learning. Legion [31] leverages Monte Carlo Tree Search (MCTS) to model the state exploration as a sequential decision-making process on the tree-structured program space. Symbolic execution is performed lazily and only when a state is deemed promising. Learch [25] trains a regression model on a set of training programs to learn the state selection policy based on a set of state-describing features. Then, the trained model is used to test unseen target programs.

In contrast, concolic execution (CE) explores the program space *iteratively*. Given an input, a CE engine (e.g., QSYM [50], SymCC [35], and SymSan [11]) executes the program concretely and simultaneously collects symbolic constraints along its concrete execution path. When a symbolic branch point (whose direction depends on the input) is encountered, the CE engine collects the constraint of the current branch condition to dictate which branch direction is taken by the concrete execution. Additionally, based on a branch-flipping policy, the CE engine may decide to generate a new input that can traverse the untaken branch direction. To do so, the CE engine constructs a constraint set that includes the negated current branch condition and a number of preceding branch conditions and queries an SMT (Satisfiability Modulo Theories) solver for a solution. Then a new input is generated by replacing parts of the original input with the values suggested by the solution. After the CE finishes processing the current input, it will pick and process one of the newly generated inputs. Obviously, the branch-flipping policy is essential for concolic execution.

2.2 Branch-Flipping Policies

The most naïve branch-flipping policy would be "flip all". As the name suggests, this policy tries to flip all possible branches. This policy can ultimately achieve the highest code coverage, given unlimited computing resources and time. No one has ever adopted this policy because computing resources and time are never unlimited, and many branches are either redundant or unworthy to be flipped.

A more realistic branch-flipping policy is to flip every branch executed through a unique execution path prefix. More specifically, this path prefix consists of a list of symbolic branches along the execution path, while concrete branches are ignored. For this reason, we refer to this policy as "PP policy". However, even with this policy, the number of branches to be flipped can still be enormous. This is because a program often contains loops and function calls, and one branch that appears in different loop iterations and different calling contexts will be flipped repeatedly due to its unique path prefix in each loop iteration and each calling context. Yun et al. observed that constraints repetitively generated by the same code are useless for finding new code coverage in real-world software [50].

Based on this observation, existing state-of-the-art CE engines (e.g., QSYM [50], SymCC [35], and SymSan [11]) adopt a more restrictive branch-flipping policy, which was first introduced in QSYM. This policy looks at branch bigrams. It will flip the current branch if its bigram (i.e., the pair of the previous symbolic branch and the current one) is new. We refer to this policy as the "BR policy" because it focuses on branches rather than paths. Compared to the PP policy, the BR policy will flip significantly fewer branches because only the last symbolic branch is included in the "context" of the current branch instead of all preceding symbolic branches.

Some CE engines explore the branch selection heuristics with respect to the branch locality. In particular, SAGE [22] executes inputs in descending order of their code coverage and only flips

[1]Named after Marco Polo, and is also short for **Mar**kov **Co**ncolic Explorer.

branches that are located below the point where the current execution trace branches off from its parent trace to avoid redundant exploration. To efficiently explore uncovered branches, CREST [6] proposes a control flow graph (CFG) directed searching algorithm to prioritize branches in close proximity to uncovered branches through the statically constructed control flow graph and call graph. Specifically, each branch is evaluated by a scalar value obtained by adding up 1) the length of the shortest path to its nearest uncovered branch and 2) the number of flipping attempts devoted to it. CREST then flips branches in ascending order of this scalar value.

2.3 Motivation

To understand the effectiveness of different branch-flipping policies, we conducted a measurement study. We equipped SymCC [35], one of the SOTA CE engines, with both PP and BR policies. We selected four programs in binutils (`objdump`, `size`, `nm-new`, and `readelf`) and assembled an input corpus of 1000 seeds for each of them. We made the following four observations.

(1) The BR policy is so overly strict that it filters out many promising branches. We investigate if the branches discarded by the BR policy are indeed useless for reaching higher code coverage. To answer this question, for each program, we compared the code coverage after SymCC processed the same input corpus and attempted to flip the branches according to the BR and PP policies, respectively. To simplify the evaluation, we did not allow SymCC to further process testcases generated from the initial input corpus. Table 1 lists the results. We can see that for all four programs, the PP policy achieved higher code coverage than the BR policy. For `readelf`, the PP policy achieved a whopping 43% higher code coverage. This evaluation shows that the BR policy can miss promising branches that could lead to higher code coverage[2].

Table 1: Code Coverage w/ Single-Pass Exploration

Program	Code Coverage	
	BR	PP
objdump	3571	4032(+13%)
size	2128	2192(+3%)
nm-new	1995	2040(+2%)
readelf	2461	3527(**+43%**)

(2) The strict BR policy often leads to early termination. Observation 1 tells us that the BR policy filters out promising branches that directly lead to new code coverage. A promising branch may *indirectly* lead to new code coverage after several generations of testcases that are derived from a testcase traversing this branch. Ideally, a good branch-flipping policy would recognize this kind of branch and make continuous progress by iteratively processing newly generated testcases. Therefore, we would like to see how well a CE engine performs when it continuously processes newly generated testcases. Table 2 presents the results of this continuous exploration under the BR policy. We can see that the CE engine terminated within five hours[3] for all four programs, because it exhausted its attempts to flip all available branches in all initial inputs and generated testcases. Moreover, the final code coverage only covers 6.53% to 13.90% of the total coverage[2].

Table 2: Code Coverage w/ Continuous Exploration

Program	Total Cov.	BR	
		Coverage	Time-to-Term.
objdump	82442	6252(7.58%)	2.68h
size	57871	3777(6.53%)	2.48h
nm-new	58378	3996(6.85%)	4.33h
readelf	31622	4394(13.90%)	**0.52h**

(3) Branches selected by the BR policy are of low quality. As described above, the BR policy uses branch bigrams to select promising branches to flip. We would expect that most of these selected branches could lead to new code coverage. Table 3 illustrates our findings on the quality of the new inputs generated from these selected branches. In fact, on average, only 27.46% of the generated inputs from the branches selected by the BR policy can lead to new coverage. In other words, the majority (72.54%) of the flipping and solving efforts do not immediately translate into code coverage gain. One major reason why *BR policy* cannot select high-quality branches is that the branch-flipping decision is *only* made based on the testcases that have been previously processed and the current testcase that is processed up to this point. It does not have a chance to examine the remaining execution of the current testcase or the remaining testcases to make a globally optimal decision.

Table 3: Quality of Generated Testcases

Program	BR	
	New-cov Testcases	Total Testcases
objdump	231(10.83%)	2,132
size	387(26.51%)	1,460
nm-new	493(45.23%)	1,090
readelf	873(27.25%)	3,204

(4) Path divergence (PD) rate of concolic execution is exceedingly high that many constraint solving efforts go wasted. We observe that oftentimes, a generated testcase does not traverse the intended unvisited path. This problem is referred to as path divergence problem [22]. Table 4 lists our findings with respect to path divergence. We can see that path divergence is very common (as high as almost 50% for `size`, and on average 28.72%). We also observe that the path divergence issue is program-specific and branch-specific. Many branches do not have path divergence at all, while other branches constantly lead to path divergence. Unfortunately, current branch-flipping policies do not take this into account, leading to the low performance of CE.

Based on the observations, we are motivated to design a new concolic execution scheme that can overcome the aforementioned limitations for more efficient testing.

[2]Edge coverage measured by SanitizerCoverage tool.
[3]All experiments conducted in this paper are measured in terms of wall time.

Table 4: Path Divergence Rate

Program	PD Count	Total Solving (excluding unsat)	PD Rate
objdump	4628	16926	27.34%
size	5304	10728	49.44%
nm-new	4668	16975	27.50%
readelf	1551	11614	13.35%
Overall	**16151**	**56243**	**28.72%**

3 DESIGN AND IMPLEMENTATION

In this section, we introduce MARCO, a novel stochastic and asynchronous concolic explorer.

Specifically, to tackle the first two limitations, unlike SymSan, MARCO keeps all path constraints from a unique path prefix and incorporates extra information, including calling context and branch direction, into branch definition to retain more meaningful branches. To address the third limitation, our system implements a reachability-guided branch scheduler that can accurately assess the potential of finding new code coverage for every branch. The scheduler then conducts asynchronous solving to make sure our decisions are globally optimal. Furthermore, to overcome the PD problem, MARCO models the PD rate for each branch and takes it into consideration when making scheduling decisions.

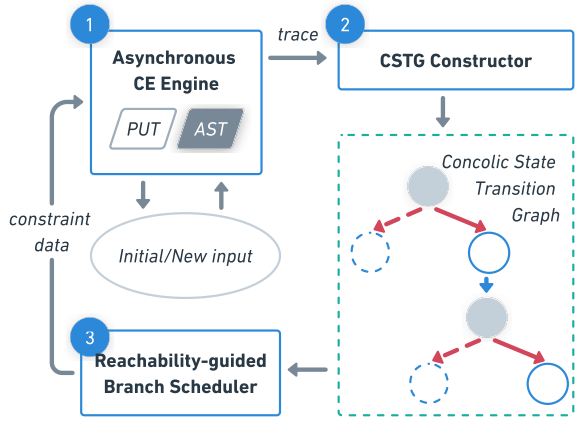

Figure 1: Approach Overview

3.1 Approach Overview

As shown in Figure 1, MARCO comprises three major components: 1) the asynchronous concolic execution engine, 2) the CSTG constructor, and 3) the reachability-guided branch scheduler.

At the beginning of the testing process, the asynchronous concolic execution engine receives an initial seed input and a binary program as input. It then performs concrete and symbolic execution simultaneously, without any constraint solving, to collect concolic traces. These traces comprise symbolic branches encountered, along with the path constraint information needed for branch flipping.

The resulting trace is then passed to the CSTG constructor, which incrementally constructs a CSTG using branch points and branches as nodes, and branch point-to-branch transitions, as well as branch-to-branch point transitions as edges. The reachability-guided state scheduler assesses the potential of each node in the CSTG, calculates a reachability score for each node, and ranks them based on their scores. The highest-ranked node has the greatest potential to lead to new code coverage. The asynchronous CE engine will be invoked to solve a path constraint from the top-ranked node for new testcase generation. MARCO then executes the testcase to collect trace and repeat the process.

3.2 A Running Example

To better explain our design, we will use an example program from [9], illustrated in Listing 1. The program takes two symbolic inputs, x and y, as input parameters for the function `testme()`. This function contains two symbolic branches located at Line 7 and 8 respectively. The directions taken at these two branches depend on the values of the symbolic inputs.

Listing 1: A Running Example

```
1  int twice (int v) {
2      return 2*v;
3  }
4
5  void testme (int x, int y) {
6      z = twice (y);
7      if (z == x) {
8          if (x > y+10) { ERROR; }
9      }
10 }
11
12 int main() {
13     x = sym_input();
14     y = sym_input();
15     testme(x, y);
16     return 0;
17 }
```

3.3 Asynchronous Concolic Execution Engine

Unlike synchronous CE engines [11, 35, 36, 50] that perform symbolic tracing and branch flipping simultaneously, MARCO takes an asynchronous approach. Specifically, it decouples branch flipping logic (which includes path condition collection and new testcase generation) from the symbolic tracing logic and defers it until after all branch points uncovered are assessed, and a global optimal branch choice is made. It is worth mentioning that although some prior works (e.g., SAGE and CREST) collect execution traces and then replay them offline for branch-flipping, the branch selection is made while processing the current trace. In other words, their branch-flipping policy adopts only a local view as compared to MARCO, which will evaluate all branches to make a global optimal selection.

Specifically, the asynchronous CE engine alternates between two modes: 1) the symbolic tracing mode, where it executes the target program with existing testcases to collect data for educated branch prioritization, and 2) the path exploration mode, where it flips a selected branch to find a new path.

3.3.1 Symbolic Tracing Mode. In this mode, the CE engine takes one Program Under Test (PUT) and one testcase as input and produces a concolic path and an AST table. Since our implementation

is based on SymSan [11], the AST table stores all the necessary information for reconstructing symbolic expressions.

A concolic path consists of a list of symbolic branches that follow the execution path and some auxiliary information. Below are related definitions:

Branch Point. A branch point bp is defined as:

$$bp = (addr, ctx) , \tag{1}$$

where $addr$ is the address of the branching instruction, and ctx represents the calling context, which is calculated as a hash of all the call sites on the call stack. The context-sensitive definition of branch point allows MARCO to differentiate a branching instruction under different calling contexts and characterize program exploration status more accurately.

For the running example, we have two branch points {L7, main → testme} and {L8, main → testme} at Line 7 and 8 respectively. For brevity, we refer to them as L7 and L8 in the following discussion.

Branch. A branch brc is defined as:

$$brc = (bp, dir) , \tag{2}$$

where bp is a branch point defined by Definition (1), and dir is the direction taken from the branch point. Each branch point has two branches. We use T and F to denote "then" and "else" branches respectively. In the running example, we have four branches denoted as L7T, L7F, L8T, and L8F.

For each symbolic branch, we need to collect essential information about its path constraints. In traditional symbolic execution, the path constraints include all preceding symbolic branches. However, this strategy is often overly strict: generating a new input that follows the exact same path and visits the untaken branch is often impossible [15]. However, there may exist a new input that follows a *slightly* different path and successfully visits the desirable branch. EXE [8] presents constraint independence optimization which divides path constraints into subsets which are dependent on disjoint sets of input bytes to solve them separately. This idea is then adopted by QSYM [50] and SymSan [11] for concolic execution. Specifically, when negating a branch, SymSan includes any preceding branch that shares data-flow dependencies with the current branch or another preceding branch already included. The resulting set of branches are referred to as *nested branches* in SymSan. Since we perform concolic execution asynchronously, we prefer not to collect branch constraints right away. Instead, we just record their nested branches.

Nested Branch Set. We define Nested Branch Set NBS for a branch brc as a set of branches in a recursive manner: if a branch brc_i has a data-flow dependency with the target branch brc, then $brc_i \in NBS(brc)$; and if $\exists brc_j \in NBC(brc)$ and brc_i has a data-flow dependency with brc_j, then $brc_i \in NBS(brc)$.

Concolic Path. A Concolic Path CP is defined as a list of 2-tuples:

$$\begin{aligned} CP = &[(brc_0, NBS(brc_0)), (brc_1, NBS(brc_1)), \\ &..., (brc_n, NBS(brc_n))] , \end{aligned} \tag{3}$$

where brc_i is the i-th symbolic branch encountered in the execution trace, and $NBS(brc_i)$ is the nested branch set of brc_i.

For the running example, there are three unique concolic paths including: {(L7F,∅)}, {(L7T, ∅), (L8F, {L7T})}, {(L7T, ∅), (L8T, {L7T})}.

Loop Pruning Optimization. Furthermore, we employ optimization to speed up the concolic path collection. Real-world programs often have many loops. Symbolic branches in loops will repeatedly appear in the concolic paths. It takes time to collect their nested branch sets, even though it is much faster than collecting the nested branch constraints. It is also unlikely to iterate through all these sets in order to generate new testcases in the later stage. Therefore, we decide to prune the nested branch sets early on. More specifically, during the execution, we trace the visit count of each encountered symbolic branch. For a branch whose visit count does not evaluate to the power of 2, we do not generate its NBS. In other words, its NBS is ∅.

In summary, in symbolic tracing mode, the CE engine traces all the testcases in the queue to collect the concolic paths and AST tables. Then it switches into path exploration mode.

3.3.2 Path Exploration Mode. In this mode, the CE engine invokes the reachability-guided branch scheduler (discussed in §3.5) to find a global optimal branch choice. With the constraint data of the chosen branch, the CE engine will assemble the path constraint set for traversing this branch and solve it to generate a new testcase.

The constraint data of the chosen branch consists of 1) brc_n, the chosen branch; 2) NBS_n, nested branch set of brc_n; and 3) the AST table of the execution where the chosen branch is encountered. We start by initializing the PC, the path constraint set as ∅. Then we query the AST table for the branch predicate of brc_n and add it into PC. If NBS_n is not empty, we query the AST table for each branch in NBS_n for their branch predicates and add them into PC. Then we reuse the solving strategy proposed in QSYM [50]. Specifically, if PC is not satisfiable, we resort to optimistic solving, which will only solve the branch predicate of the target branch and disregard any predicates collected from NBS_n. If optimistic solving is not viable either, the branch scheduler will be prompted again to generate another set of constraint data until a new seed is generated.

3.4 CSTG Constructor

After collecting concolic paths in the asynchronous CE engine, we seek to construct CSTG, which further enables the reachability-guided branch scheduler. The graph is a directed heterogeneous graph defined as follows.

Concolic State Transition Graph (CSTG). A CSTG is defined over a set of CPs as:

$$G = (V, E) , \tag{4}$$

where V is a set of branch points and branches, and E is a set of vertex transitions. In addition, a virtual root vertex denotes the program entry point. Each vertex $v \in V$ is associated with a set of attributes including: 1) $v.vis$: the number of concrete visits at v; 2) $v.atp$: the number of branch flipping attempts at v; 3) $v.win$: the number of successful branch flipping attempts at v; and 4) $v.pcq$: the queue of path constraint sets for generating new testcase that potentially will traverse v. Note that attributes 2) to 4) only apply to vertices representing branches instead of branch points. Each edge $e \in E$ is associated with a concrete visit count $e.vis$.

Algorithm 1 illustrates how MARCO constructs the CSTG incrementally. Initially, the graph contains one root node R as the program entry, and the edge set is empty. The procedure takes as

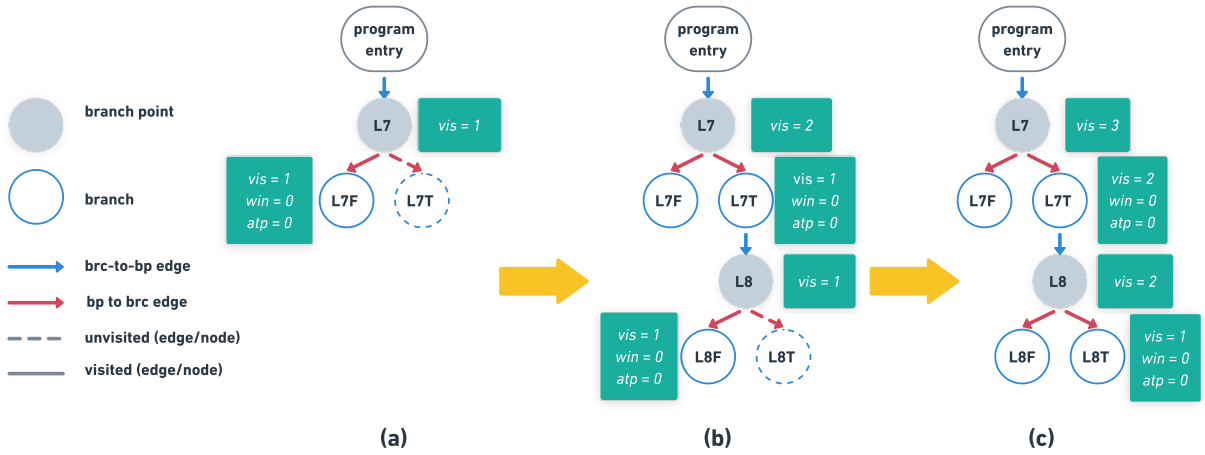

Figure 2: CSTG Construction of Example Program

Algorithm 1 The CSTG Construction Algorithm

```
1:  lastChosen ← state chosen from last scheduling round
2:  procedure GraphUpdate(G, CP, lastChosen)
3:      lastnode = R
4:      while ! CP.empty() do
5:          (addr, ctx, dir) = CP.pop()
6:          if !G.findNode(addr, ctx) then
7:              bp = G.newNode(addr, ctx)
8:              brc0 = G.newNode(addr, ctx, dir)
9:              brc1 = G.newNode(addr, ctx, !dir)
10:             G.newEdge(<lastnode, bp>,<bp, dp0>,<bp, dp1>)
11:         else
12:             bp = G.getNode(addr, ctx)
13:             brc0 = G.getNode(addr, ctx, dir)
14:             brc1 = G.getNode(addr, ctx, !dir)
15:         end if
16:         for s ∈ [bp, brc0, brc1] do
17:             G.updateNode(s, lastChosen)
18:         end for
19:         for e ∈ [<lastnode, bp>,<bp, brc0>] do
20:             e.vis++
21:         end for
22:         lastnode = brc0
23:     end while
24: end procedure
```

input the graph G and a new concolic path CP as defined in (3). The algorithm then performs a preprocessing step to retain a set of visited branches along with their concrete path prefixes and remove from CP any branch that is visited through the observed path prefix. When a new branch (according to the Definition 2) is observed for the first time, we insert three nodes (one for its branch point bp and two for the taken and untaken branches brc_0 and brc_1, and three edges into the current graph (Ln.6-10). If the branch has been observed and thus has already existed in the graph, we simply retrieve the existing three nodes (one branch point and two branches) from the graph (Ln.12-14). Then, the algorithm calls *updateNode* to update the nodes' attributes defined in §3.4 as needed (Ln.16-17). Specifically, for bp and brc_0, we update visit count $v.vis$. If brc_0 matches *lastChosen*, we update its win count $v.win$. For brc_1, we update the path constraint queue $v.pcq$ to include the new path constraint collected from the current execution path to potentially force execution down brc_1. Further, if the currently taken branch brc_0 is equal to the node picked by the last scheduling round to perform branch flipping on *lastChosen*, it means the testcase generated from the last round (i.e., the current testcase) indeed traverses the selected branch. In this case, it will increase the current branch's win count by one.

In our running example, we consider three concolic paths {(L7F,∅)}, {(L7T, ∅), (L8F, {L7T})}, {(L7T, ∅), (L8T, {L7T})}. MARCO gradually constructs CSTG of the example program as illustrated in Figure 2 (a), (b), and (c).

Initially, the graph is empty with a root node, which denotes the program entry. For the first concolic path {Ł7F}, since node L7F is a new node, we add three nodes, i.e. L7, L7F, and L7T, and three edges, i.e. (R, L7), (L7, L7F), (L7, L7T), into the graph. We increase the visit counts of node L7, L7F, edge (R, L7), (L7, L7F) from 0 to 1. After processing this concolic path, the graph is presented as Figure 2(a). Similarly, MARCO then takes the second concolic path {L7T, L8F} as input. As the first branch L7T already exists in the graph, we increase the visit counts of node L7, L7T and edge (L7, L7T) by 1. However, the second branch L8F is not an existing node in the graph. Therefore, we insert three nodes, i.e. L8, L8F and L8T, and three edges (L7T, L8), (L8, L8F), (L8, L8F) into the graph. And we update the visit counts accordingly. After processing this concolic path, the graph is shown as Figure 2(b). Moreover, we make similar changes as discussed above for the concolic path {L7T, L8T}. Hence, after processing the three concolic paths, Figure 2(c) is the final CSTG, which will then be used for branch scheduling.

3.5 Reachability-guided Branch Scheduler

Reachability-based branch scheduler aims to find the branch that bears the highest potential for new code coverage and gives the path constraint data of the top-ranked node to the asynchronous CE engine for input generation.

Essentially, we assess the potential of a branch by the number of reachable yet unvisited branches deeper in the execution paths that

traverse the branch. To do so, we generate a reward score for each untaken branch, consider a concolic trace as Markov Chain and compute a transition probability to take the path divergence (PD) rate into consideration, and further accumulate the rewards up to calculate a node reachability score that estimates the potential of every branch in the CSTG, in order to pick the best one for further exploration. However, one technical challenge is that the transition probability between nodes and the estimated reward of nodes are unknown at the beginning of the testing. Here in this section, we discuss how MARCO tackles this challenge.

Edge Transition Probability Calculation. The transition probability of an edge captures how likely an execution will branch to the end node from the start node. As discussed in §3.4, there are two types of edges in MARCO: 1) the *bp*-to-*brc* edges and 2) the *brc*-to-*bp* edges. And we calculate their transition probability differently.

The transition probability of a *bp*-to-*brc* edge is associated with the success rate of generating a testcase traversing *brc* by solving a path constraint associated with *brc*, i.e., the opposite of the path divergence rate of this edge. The total solving attempt count at *brc* is $brc.atp$, and the success count is $brc.win$. Intuitively, the estimated transition probability is $brc.win/brc.atp$. The estimation is relatively accurate when the attempt count at *brc* is high enough. But this assumption does not always hold, especially at the early stage of testing and for the less-explored code regions. For better estimation of the transition probability and to balance between exploration and exploitation, we resort to *Thompson Sampling* (TS) [39]. The key idea of TS is to sample the success rate of an action over the *Beta Distribution* defined by the outcomes of the past trials. The Beta distribution is defined by two positive parameters α, denoting the win count, and β, denoting the loss count. It becomes more and more concentrated around the empirical success rate $\alpha/(\alpha+\beta)$ as the number of total trials $(\alpha+\beta)$ grows. For a *bp*-to-*brc* edge, α is $brc.win$ and β is ($brc.atp$ - $brc.win$). The transition probability of a *bp*-to-*brc* edge is calculated by Equation 5.

$$p(bp, brc) = \tau(y.win, y.atp - y.win)\ , \tag{5}$$

where τ denotes Thompson Sampling. Note that, each *bp* has two *bp*-to-*brc* edges, each leading to one viable branch. We normalize the transition probabilities of these two edges to ensure they sum up to one.

The *brc*-to-*bp* edge transition differs from that of *bp*-to-*brc* edge in the following aspects: 1) CE engine cannot actively steer execution from a *brc* node to one particular *bp* node through path constraint solving; 2) one *bp* node has two outgoing edges, each leading to one viable branch, while a *brc* node potentially has zero to multiple succeeding *bp* nodes; 3) each *brc* has only one parent node which is its branch point while each *bp* can potentially have multiple preceding *brc* nodes. Therefore, Equation 5 does not apply to computing the transition probability for a *brc*-to-*bp* edge.

The transition probability of an edge leading from *brc* to *bp* can be estimated as the success rate of transitioning from *brc* to *bp*. In this case, each visit at edge $e_{brc,bp}$ is considered a win. The total amount of trials for visiting this edge includes the visit count and attempt count at *brc*. In other words, each time *brc* is visited or attempted, but the subsequent execution does not lead from *brc* to *bp* is considered a losing attempt. Similarly, when the total trial count is low, the accuracy of the estimation can be low. Again, we leverage TS for the transition probability computation for *brc*-to-*bp* edge with α being $e_{brc,bp}.vis$ and β being $brc.vis + brc.atp - e_{brc,bp}.vis$ as shown in Equation 6.

$$p(brc, bp) = \tau(e_{brc,bp}.vis, brc.vis + brc.atp - e_{brc,bp}.vis) \tag{6}$$

Again, we normalize the transition probabilities of the edges leading from the same *brc* node and ensure they sum up to one.

In summary, we leverage Thompson Sampling to dynamically optimize the estimation of transition probabilities of the two types of edges in CSTG which allows us to balance between exploration and exploitation.

Node Reachability Score Calculation. We compute a node reachability score for each node in CSTG, which indicates the nodes' potential for leading to new code coverage in future testing. We then use it to guide the path prioritization in concolic execution.

The reachability score of the *brc* node should capture two aspects: 1) potential new code coverage reachable from *brc* and 2) the difficulty of generating a new testcase that visits the *brc* node. We measure a node's reachable new code coverage as a numerical value denoted as *Coverage Score* and compute it by Equation 7 (for leaf nodes) and Equation 8 (for interior nodes).

$$N.score = \tau(0, N.vis + N.atp) \tag{7}$$

The coverage score of a leaf node is calculated by Equation 7. In particular, for an unvisited leaf node, the coverage score is affected by the number of solving attempts devoted to it. When the number of attempts grows but the node remains unvisited, it means that this node could be too hard to reach. Therefore, our limited resources are better off being relocated to other nodes. For a visited leaf node, apart from the attempt count, the visit count also affects its coverage score. Each visit to a node without steering the execution into a deeper state is considered a failed attempt. Consequently, the exploration should try to avoid such nodes. As the visit count and the attempt count grow, the coverage score of a visited leaf node will decrease.

We compute an interior node's coverage score by Equation 8:

$$N.score = \sum_{i=0}^{n} p(N, M_i) * M_i.score\ , \tag{8}$$

where M_i ($i \in [0, n]$) denotes a child node of N. Essentially, the coverage score of an interior node is affected by two major factors. First, a node that is adjacent to a large number of unvisited nodes is in general of higher potential than a node that has only a small number of unvisited neighbor nodes. Hence, the number of a node's unvisited successors in CSTG can strongly indicate its potential for new code coverage. Second, given any path in a program, the number of inputs that go through the child node is strictly less than or equal to the number of inputs that go through the parent node. Subsequently, the distance between a node and its unvisited successors also plays an essential role in estimating the potential for new coverage.

The coverage score of each node in CSTG is updated periodically to reflect the most recent changes. Apparently, CSTG can be a cyclic

graph which imposes a challenge for efficiently updating each one of the nodes for an updated coverage score. We periodically perform the whole graph score updates by first performing a post-order traversal over the graph to extract all the nodes into an ordered list. Then we traverse the list to update each node's coverage score. The reachability score for each *brc* node in the graph is computed by Equation 9.

$$brc.rs = p(bp, brc) * brc.score \tag{9}$$

Essentially, for a branch *brc* with a high path divergence rate (i.e. low in-edge transition probability), it is hard to generate a testcase traversing that branch and it renders the coverage score in vain. We then prioritize the branch nodes in CSTG for scheduling by their reachability score.

Branch Prioritization. After calculating reachability scores, the node (i.e., branch) with a better potential of reaching new code will have a higher score and be promoted in the scheduling. The path constraint associated with this top-ranked node is sent to the Path Constraint Solver for new input generation. In case of an unsatisfiable path constraint, the scheduler is prompted again until a new testcase is successfully generated and the testing continues.

4 EVALUATION

In this section, we evaluate the efficacy of our proposed approach by answering the following research questions:

- **RQ1: Effectiveness of end-to-end concolic execution.** Can our model improve the performance of end-to-end concolic execution?
- **RQ2: Effectiveness of design choices.** What are the unique contributions of each design choice in MARCO?
- **RQ3: Vulnerability detection.** Can MARCO be more effective when detecting vulnerabilities?

4.1 Evaluation Plan

To better answer the aforementioned research questions, we use the following configurations:

- **SymSan [11]**, the SOTA CE engine, which adopts the traditional synchronous solving (i.e., the constraint solving is conducted at the time when the branch is encountered) with the same native branch flipping policy as QSYM [50]. This baseline is to directly compare with MARCO.
- **SymSan-pp**, a variant of SymSan that adopts a *PP* branch-flipping policy, where each branch is defined by its path prefix. The testcases are executed in a First-In-First-Out (FIFO) order, with all branches flipped by the visit order. This baseline is to show that simply selecting more branches to flip will not improve CE performance.
- **MARCO-rdm**, a variant of MARCO that defines each branch by its path prefix and picks a random branch from the last visited program path to flip and generate a new testcase. This configuration is to demonstrate the effectiveness of our branch scheduling strategy.
- **MARCO-cfg**, a configuration that applies the CFG-directed searching algorithm of CREST [6] on our dynamically generated CSTG. The assessment of each branch is determined by the branch count between the branch itself and the nearest unvisited branch, as well as the flipping attempt count. This configuration is used to show the effectiveness of our branch scheduling strategy.
- **MARCO-uv**, a variant of MARCO which only allows the scheduler to pick from *unvisited* nodes. This configuration is used to show the necessity of flipping the visited branches.
- **MARCO-MC**, a variant of MARCO with Markov Chain modeling but no Thompson Sampling. This configuration evaluates the importance of Thompson Sampling in MARCO.
- **MARCO**, our full-fledged system.

To answer RQ1, we compare the code coverage metric of the full-fledged model against SymSan. The experiment is conducted on real-world programs listed in Table 5. For RQ2, we compare the code coverage per path constraint solving for all the configurations listed to showcase the effectiveness of each design choice. Finally, to answer RQ3, we run both MARCO and SymSan on the UniFuzz dataset, and compare the number of unique bugs found by them.

Dataset. We collect a dataset consisting of 30 popular real-world programs as shown in Table 5, as well as 71 programs from the DARPA Cyber Grand Challenge (CGC) dataset. To answer RQ1 and RQ2, we conduct experiments on the CGC programs, as well as programs No. 1 to 16 (Binutils and Fuzzbench [1] binaries). To answer RQ3, we further leverage programs No. 17 to 30, which are the subset of the MARCO compatible Unibench dataset proposed in UniFuzz [29]. We configure the experiment to align with the original setup in UniFuzz, including each program's execution option and the initial seed corpus used.

Table 5: Details of Real-world Applications Evaluated

No.	Program	Version	No.	Program	Version
1	nm-new	2.33.1	16	libtiff	2e822691
2	readelf	2.33.1	17	tcpdump	4.8.1 + libpcap 1.8.1
3	objdump	2.33.1	18	flvmeta	1.2.1
4	size	2.33.1	19	tiffsplit	libtiff 3.9.7
5	libpng	1.2.56	20	jhead	3.00
6	libxml2	2.9.2	21	imginfo	jasper 2.0.12
7	file	5.42	22	jq	1.5
8	vorbis	c1c2831f	23	lame	lame 3.99.5
9	curl	2481dbe	24	wav2swf	swftools 0.9.2
10	lcms	430f916	25	mujs	1.0.2
11	woff2	9476664	26	sqlite3	3.8.9
12	libjpeg-turbo	b0971e47	27	mp3gain	1.5.2-r2
13	sqlite3	c78cbf2	28	mp42aac	Bento3 1.5.1-628
14	tcpdump	4.99.1	29	cflow	1.6
15	freetype	cd02d359a	30	infotocap	ncurses 6.1

Experiment Setup. All evaluation was done on a workstation with two-socket, 48-core, 96-thread Intel Xeon Platinum 8168 processors. The workstation has 768G memory. The operating system is Ubuntu 18.04 with kernel 5.4.0.

4.2 RQ1: Effectiveness of MARCO

To demonstrate the effectiveness of MARCO in terms of exploring new code coverage, we measure the edge coverage during testing and compare our full-fledged model with SymSan. Each configuration is repeated 10 times to reduce randomness.

We collect the edge coverage at the end of each 24h trial and measure the coverage improvement ratio of MARCO over the baseline SymSan. For each program, we further investigate the relative code coverage between SymSan and MARCO with the formula proposed

in QSYM [50]. For code coverage A (MARCO) and B (SymSan), we can quantify the coverage difference by using:

$$d(A,B) = \begin{cases} \frac{|A-B|-|B-A|}{(A\cup B)-(A\cap B)} & \text{if A} \neq \text{B} \\ 0 & \text{otherwise} \end{cases} \quad (10)$$

With the coverage difference score $d(A,B)$, we can infer the number of unique edges that A covered, out of the total edge coverage that either A or B can uniquely explore. A positive score means A (MARCO) finds more unique coverage than B (SymSan). The value will be 1 if A (MARCO) not only finds more coverage than B (SymSan) but also covers all edge coverage explored by B (SymSan).

In our experiment, we evaluate the performance of MARCO on 16 real-world programs (programs No.1 to 16 in Table 5). On average, MARCO is able to cover 13.03% more edges, with a maximum improvement on `readelf` for 88.56%. This indicates the effectiveness of our approach in improving the effectiveness of concolic testing for real-world programs. Out of the 16 tested programs, MARCO finds more edge coverage than SymSan on 11 programs (68.75%). Moreover, MARCO dominates the coverage findings on three targets (`file`, `lcms`, and `sqlite3`), where it also covers all the edges found by SymSan.

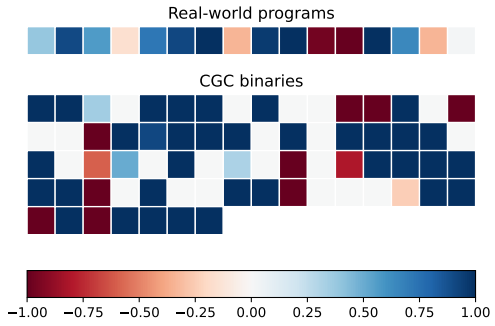

Figure 3: Coverage Difference Score of Real-world Programs and CGC Binaries

We further evaluate the performance of MARCO on 87 programs (71 DARPA CGC binaries + 16 real-world programs) and compute the coverage difference score between MARCO and SymSan. Inspired by [50], we visualize the results in Figure 3: the blue color indicates that MARCO finds more edge coverage than SymSan, and the red color indicates that SymSan finds more. The results indicate that MARCO can do better than SymSan on 49 programs, and worse on 17 programs.

Further investigation (Table 6) shows that SymSan would terminate within 5 hours on 75% (12/16) of the real-world programs, even though there still exist many edges unexplored. This is due to the overly strict branch definition and ill-advised branch-flipping strategy adopted in SymSan.

To evaluate the scalability of MARCO, we investigate the graph size growth and the memory cost for each of the 16 real-world programs during testing. The results show that the number of nodes in CSTG grows sub-linearly during the 24h trials. At the end of each trial, the minimum, maximum, average, and median values of the node counts are 0.26k, 118.98k, 13.89k, and 3.10k correspondingly. We then record the amount of memory taken by storing the AST tables and see that the disk usage grows linearly. At the end of the trial, the minimum, maximum, average, and median values of memory usage are 7.88G, 63.19G, 23.90G, and 20.66G.

Table 6: Average Termination Time for SymSan

Program	Term. Time(h)	Program	Term. Time(h)
nm-new	4.33±1.40	curl	0.10±0.03
readelf	0.52±0.05	lcms	0.06±0.01
objdump	2.68±0.54	woff2	>24
size	2.48±0.31	libjpeg-turbo	15.43±2.90
libpng	0.05±0.01	sqlite3	0.02±0.00
libxml2	1.57±0.16	tcpdump	0.75±0.81
file	0.27±0.01	freetype	10.36±0.79
vorbis	8.66±0.79	libtiff	1.93±0.26

4.3 RQ2: Effectiveness of Design Choices

As discussed earlier, SymSan terminates very early in 75% of the tested programs, meaning only a limited number of solving attempts have been made. In RQ2, we allocate the same amount of solving attempts for the other configurations and assess their new code coverage. By doing so, we can demonstrate the effectiveness of our design choices in improving the branch prioritization scheme.

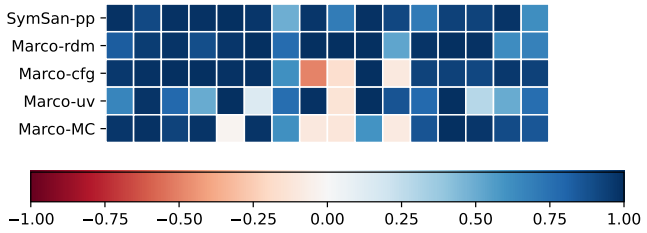

Figure 4: Coverage Difference Score Within Solving Budget

We look into the edge coverage for SymSan-pp, MARCO-rdm, MARCO-cfg, MARCO-uv, and MARCO-MC with the 16 real-world programs and compute the coverage difference scores compared with MARCO as defined in Equation 10. The experimental results are displayed in Figure 4. Each row depicts the coverage difference score of A (MARCO) and B (the baseline labeled to the left of the row). The blue color indicates that MARCO finds more edge coverage than the corresponding baseline, and the red color suggests otherwise. The results exhibit a few major conclusions:

Firstly, MARCO outperforms SymSan-pp and MARCO-rdm on all 16 programs. On average, MARCO covers 55.99% and 86.64% more code than SymSan-pp and MARCO-rdm. This result explicitly demonstrates that the novel branch prioritization strategy in MARCO, other than a simple FIFO or random selection, is extremely useful when it comes to code exploration.

Secondly, MARCO outperforms MARCO-cfg on 13 out of 16 tested programs. For the other three programs (`vorbis`, `curl`, and `woff2`) where MARCO-cfg finds more coverage, the differences are slight (<1%). MARCO is able to find 83.92% more code coverage than MARCO-cfg. This result indicates that our branch flipping strategy is better than the CFG-directed approach.

Thirdly, compared with MARCO-uv, MARCO manages to find more coverage on 15 programs out of 16, with only one exception `curl`.

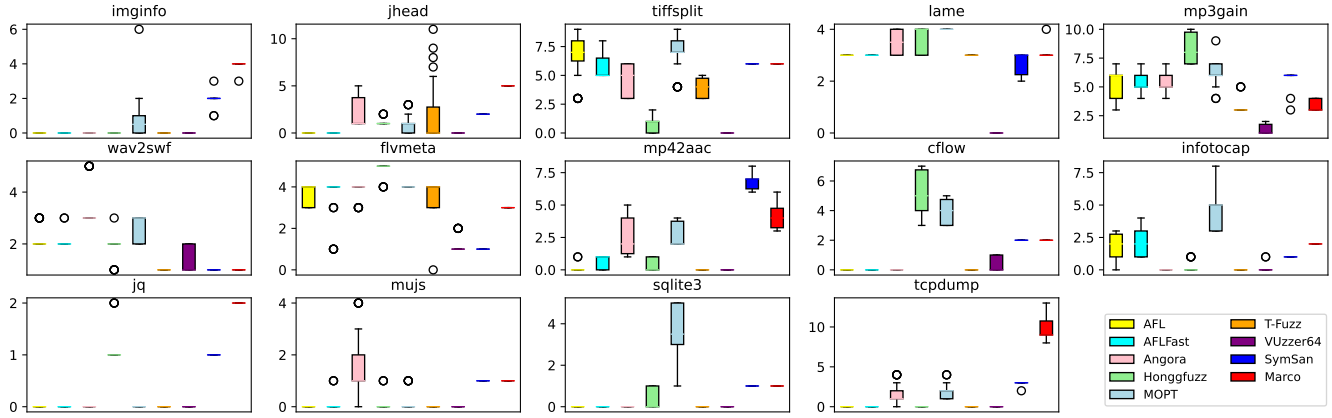

Figure 5: Number of Unique Bugs Detected

On average, MARCO finds 29.22% more code coverage than MARCO-uv. This shows that it is indeed a good strategy to deem both visited and unvisited nodes as candidates for path constraint solving.

Lastly, MARCO outperforms MARCO-MC on 12 out of 16 tested programs with an average coverage improvement ratio of 57.21%, indicating that modeling edge transition and reachability score with Thompson Sampling to balance between exploration and exploitation is crucial to making effective branch prioritization decisions.

We further evaluate the significance of difference comparing MARCO and the other configurations in Figure 4 across the 16 tested programs on their code coverage findings using p-values from the `Mann-Whitney U-Test`. We use p-value < 0.05 as the threshold for statistical significance. We observed p-values above 0.05 in only two programs, `curl` (0.07) and `woff2` (0.48), when comparing MARCO and MARCO-cfg. For the rest of the results, the p-values are below 0.001 for majority of the cases. The result suggests significant difference between MARCO and the other configurations.

4.4 RQ3: Vulnerability Detection

Lastly, we showcase the capability of vulnerability detection for MARCO by using the UniFuzz dataset, which consists of 14 programs. Specifically, we run both MARCO and SymSan 5 times for 24 hours and compare the average number of unique bugs detected. According to the results, MARCO is able to find 33.52% more bugs (47.8 v.s. 35.8) than SymSan. Among them, MARCO can uniquely identify 2.41 times the bug count of SymSan (20.5 v.s. 8.5). More concretely, MARCO finds more unique bugs than Symsan on 7 programs, less on 2, and the same amount on 5. These numbers show that MARCO has its unique advantages when finding vulnerabilities compared with state-of-the-art CE engines.

We further cross-check our results with that reported in UniFuzz paper[4] in UniFuzz [29] paper for 7 fuzzers (AFL [51], AFLFast [5], Angora [13], HonggFuzz [48], MOPT [32], T-Fuzz [34] as well as VUzzer64 [37]) in 24h. We draw the box plot of all 8 baselines in Figure 5. According to the result, MARCO is able to find more unique bugs in 12h than any of the 7 fuzzers can find in the 24h trial on 5 (`imginfo`, `jhead`, `mp42aac`, `jq` and `tcpdump`) out of the 14 tested programs. MARCO ranks the second place on `mujs` and `sqlite3`, second to Angora and MOPT respectively. We further explore the statistical rankings among MARCO and the 7 fuzzers by their average unique bug detection counts for each program. MARCO beats 6 fuzzers and is second to MOPT only. This demonstrates that MARCO can find bugs very efficiently.

[4]We contacted the authors for the original experiment data but didn't get a response by the time of submission. Therefore we repopulated the bug detection result based on the data reported in their supplementary result: https://github.com/unifuzz/supplementary_results/blob/master/UNIFUZZ_Supplementary_Paper.pdf

5 DISCUSSION

MARCO still has some limitations. First, MARCO requires access to the source code for instrumenting the PUT with the symbolic tracing and branch-solving logic through the compiling pass. In practice, however, a real-world PUT can be developed with a set of external libraries whose source codes are not accessible. In addition, the cost of reconstructing the branch dependency for recovering the nested path constraint set can be expensive when the dependency is very complex. We leave the performance optimization as future work. Moreover, in this paper, we do not study how to coordinate concolic execution with fuzzing better. SymSan [14] reported that the existing hybrid fuzzing scheme cannot consistently outperform pure fuzzing such as AFL++ [18]. A recent work by Jiang et al. [27] proposed edge-oriented scheduling to improve the performance of hybrid fuzzing. How a more efficient and intelligent concolic execution engine affects the design of hybrid fuzzing deserves more investigation.

6 RELATED WORK

In this section, we discuss precedent works closely related to MARCO.

Symbolic Execution. Symbolic Execution [21, 40] has proven to be a powerful program testing technique for test case generation and bug detection. However, due to the state explosion problem and the large overhead imposed by path constraint solving, it has low scalability for real-world program testing. To tackle this problem, a series of research has been done to prune states [7, 22, 43, 49], merge states [24, 28, 41], prioritize states [25, 31], and perform constraints reduction [16] and solution caching [7, 8] in order to improve the scalability.

Seed Scheduling in Fuzzing. Many techniques [4, 10, 20, 23, 30, 33, 45, 47, 52, 53] have been proposed for improving fuzzing. One important optimization is to improve the seed selection [26, 38]. AFLfast [5] favors the less explored paths and allocates more powers for them. Vuzzer [37] prioritizes test cases that traverse paths, which are more likely to reveal a vulnerability. Entropic [3] leverages information-theoretic entropy for scheduling seeds to optimize coverage gain and bug-finding ability. AFL-Hier [46] implements a reinforcement learning model for scheduling seeds clustered by multi-level coverage metrics. K-scheduler [42] performs graph centrality analysis to promote seeds that have a higher potential for reaching new code coverage.

Markov Chain Modeling. In AFLfast [5], the greybox fuzzing process is modeled as Markov Chain to identify the high-frequency paths and steer exploration away from those paths. Sparks et al. [44] model the program control flow as Markov Chain and seek to drive the testing toward less-explored code regions leveraging a fitness function concerning the path exploration frequency. Other reinforcement learning strategies such as Probably Approximately Accurate (PAC) bounds [17, 19] were proposed for solving Markov Decision Process. Integrating these approaches remains a potential future direction of this paper.

7 CONCLUSION

In this work, we propose to model the concolic execution as a Markov Chain process and construct a stochastic concolic state transition graph to assess each branch's potential for code coverage in a global view. The states are evaluated by their reachability to new code coverage with respect to path divergence rate along the execution trace. Evaluation of our prototype MARCO shows that the new approach proposed in this paper outperforms the state-of-the-art concolic execution engine in both code coverage and vulnerability detection.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their helpful and constructive comments. This work was supported by NSF under grant No. 2133487.

REFERENCES

[1] 2022. FuzzBench: Fuzzer Benchmarking As a Service. https://google.github.io/fuzzbench.
[2] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.
[3] Marcel Böhme, Valentin JM Manès, and Sang Kil Cha. 2020. Boosting fuzzer efficiency: An information theoretic perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 678–689.
[4] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2329–2344.
[5] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1032–1043.
[6] Jacob Burnim and Koushik Sen. 2008. Heuristics for scalable dynamic test generation. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 443–446.
[7] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *8th USENIX Symposium on Operating Systems Design and Implementation*, Vol. 8. 209–224.
[8] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. 2008. EXE: Automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)* 12, 2 (2008), 1–38.
[9] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90.
[10] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. 2020. MUZZ: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In *29th USENIX Security Symposium (USENIX Security 20)*. 2325–2342.
[11] Ju Chen, Wookhyun Han, Mingjun Yin, Haochen Zeng, Chengyu Song, Byoungyoung Lee, Heng Yin, and Insik Shin. 2022. SYMSAN: Time and Space Efficient Concolic Execution via Dynamic Data-flow Analysis. In *31st USENIX Security Symposium (USENIX Security 22)*. 2531–2548.
[12] Ju Chen, Jinghan Wang, Chengyu Song, and Heng Yin. 2022. JIGSAW: Efficient and Scalable Path Constraints Fuzzing. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 1531–1531.
[13] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 711–725.
[14] Ting Chen, Xiaodong Lin, Jin Huang, Abel Bacchus, and Xiaosong Zhang. 2015. An empirical investigation into path divergences for concolic execution using CREST. *Security and Communication Networks* 8, 18 (2015), 3667–3681.
[15] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. *Acm Sigplan Notices* 46, 3 (2011), 265–278.
[16] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2012. The S2E platform: Design, implementation, and applications. *ACM Transactions on Computer Systems (TOCS)* 30, 1 (2012), 1–49.
[17] Eyal Even-Dar, Shie Mannor, and Yishay Mansour. 2002. PAC bounds for multi-armed bandit and Markov decision processes. In *Computational Learning Theory: 15th Annual Conference on Computational Learning Theory, COLT 2002 Sydney, Australia, July 8–10, 2002 Proceedings 15*. Springer, 255–270.
[18] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies*. 10–10.
[19] Jie Fu and Ufuk Topcu. 2014. Probably approximately correct MDP learning and control with temporal logic constraints. *arXiv preprint arXiv:1404.7073* (2014).
[20] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 679–696.
[21] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 213–223.
[22] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. 2008. Automated whitebox fuzz testing.. In *Network and Distributed System Security Symposium, NDSS*, Vol. 8.
[23] Wookhyun Han, Byunggill Joe, Byoungyoung Lee, Chengyu Song, and Insik Shin. 2018. Enhancing memory error detection for large-scale applications and fuzz testing. In *Network and Distributed Systems Security (NDSS) Symposium 2018*.
[24] Trevor Hansen, Peter Schachte, and Harald Søndergaard. 2009. State joining and splitting for the symbolic execution of binaries. In *Runtime Verification: 9th International Workshop, RV 2009, Grenoble, France, June 26-28, 2009. Selected Papers 9*. Springer, 76–92.
[25] Jingxuan He, Gishor Sivanrupan, Petar Tsankov, and Martin Vechev. 2021. Learning to explore paths for symbolic execution. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2526–2540.
[26] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L Hosking. 2021. Seed selection for successful fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 230–243.
[27] Ling Jiang, Hengchen Yuan, Mingyuan Wu, Lingming Zhang, and Yuqun Zhang. 2023. Evaluating and Improving Hybrid Fuzzing. In *Proceedings of the 45th International Conference on Software Engineering, ser. ICSE*, Vol. 23.
[28] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient state merging in symbolic execution. *Acm Sigplan Notices* 47, 6 (2012), 193–204.
[29] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, et al. 2021. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers.. In *USENIX Security Symposium*. 2777–2794.
[30] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. 2022. PATA: Fuzzing with path aware taint analysis. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–17.
[31] Dongge Liu, Gidon Ernst, Toby Murray, and Benjamin IP Rubinstein. 2020. Legion: Best-first concolic testing. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 54–65.
[32] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers.. In *USENIX Security Symposium*. 1949–1966.

[33] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. Parmesan: Sanitizer-guided greybox fuzzing. In *Proceedings of the 29th USENIX Conference on Security Symposium*. 2289–2306.

[34] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 697–710.

[35] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *29th USENIX Security Symposium (USENIX Security 20)*. 181–198.

[36] Sebastian Poeplau and Aurélien Francillon. 2021. SymQEMU: Compilation-based symbolic execution for binaries.. In *Network and Distributed System Security Symposium, NDSS*.

[37] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. Vuzzer: Application-aware evolutionary fuzzing.. In *Network and Distributed System Security Symposium, NDSS*, Vol. 17. 1–14.

[38] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing seed selection for fuzzing. In *23rd USENIX Security Symposium (USENIX Security 14)*. 861–875.

[39] Daniel J Russo, Benjamin Van Roy, Abbas Kazerouni, Ian Osband, Zheng Wen, et al. 2018. A tutorial on thompson sampling. *Foundations and Trends® in Machine Learning* 11, 1 (2018), 1–96.

[40] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A concolic unit testing engine for C. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 263–272.

[41] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. 2015. Multise: Multi-path symbolic execution using value summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 842–853.

[42] Dongdong She, Abhishek Shah, and Suman Jana. 2022. Effective seed scheduling for fuzzing with graph centrality analysis. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2194–2211.

[43] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *Network and Distributed System Security Symposium, NDSS*, Vol. 1. 1–1.

[44] Sherri Sparks, Shawn Embleton, Ryan Cunningham, and Cliff Zou. 2007. Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. IEEE, 477–486.

[45] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 579–594.

[46] Jinghan Wang, Chengyu Song, and Heng Yin. 2021. Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing. In *Network and Distributed Systems Security (NDSS) Symposium*.

[47] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yuqun Zhang, Guowei Yang, Huixin Ma, Sen Nie, Shi Wu, Heming Cui, and Lingming Zhang. 2022. Evaluating and improving neural program-smoothing-based fuzzing. In *Proceedings of the 44th International Conference on Software Engineering*. 847–858.

[48] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2313–2328.

[49] Qiuping Yi, Zijiang Yang, Shengjian Guo, Chao Wang, Jian Liu, and Chen Zhao. 2017. Eliminating path redundancy via postconditioned symbolic execution. *IEEE Transactions on Software Engineering* 44, 1 (2017), 25–43.

[50] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Security Symposium (Security)*. Baltimore, MD.

[51] M. Zalewski. [n. d.]. American fuzzy lop. http://lcamtuf.coredump.cx/afl/.

[52] G Zhang, P Wang, T Yue, X Kong, S Huang, X Zhou, and K Lu. 2022. Mobfuzz: Adaptive multi-objective optimization in gray-box fuzzing. In *Network and Distributed Systems Security (NDSS) Symposium*, Vol. 2022.

[53] Xiaogang Zhu and Marcel Böhme. 2021. Regression greybox fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2169–2182.