Dynamic Path Pruning in Symbolic Execution

Ying-Shen Chen*, Wei-Ning Chen*, Che-Yu Wu*, Hsu-Chun Hsiao*†, and Shih-Kun Huang‡
*Department of Computer Science and Information Engineering, National Taiwan University, Taiwan
†Research Center for IT Innovation, Academia Sinica, Taiwan
‡Department of Computer Science, National Chiao Tung University, Taiwan

***Abstract*—To alleviate path explosion in symbolic execution, *path pruning* removes unsatisfiable paths at an early stage before they multiply. Although existing symbolic execution platforms have implemented several path pruning strategies to determine whether and when to check a path's satisfiability, it remains unclear how effective these strategies are because the time to check a path's satisfiability is non-negligible and may vary drastically. This work proposes *dynamic path pruning (DPP)*, a strategy that aims to minimize the overall exploration time by dynamically adjusting the path checking rate. DPP assigns a higher checking rate to paths that are more likely to be unsatisfiable, and the likelihood is estimated based on the observed program's characteristics, such as the observed percentage of satisfiable paths. DPP is implemented on top of an open source symbolic execution platform in only a few hundred lines. Our evaluation confirms that DPP consistently achieves near-optimal exploration time for a wide spectrum of programs, whereas existing static path pruning strategies suffer from unacceptable worst-case performance due to their program-independent behaviors: Compared with static strategies, DPP performs best in 84% (110 out of 131) of CGC binaries, and the exploration time is within 100–124% of the best static strategy in 95% of the tested hand-crafted and coreutils binaries.**

I. Introduction

With the rapid growth of software applications, manual program analysis has become inadequate. Automated program analysis is required to keep up with growth and to find vulnerabilities in a timely manner. One promising automatic analysis technique is *symbolic execution*, which can systematically explore all execution paths[1] in a program, and therefore can automatically discover inputs that trigger bugs or specific instructions. Symbolic execution uses symbolic variables rather than concrete values to simulate a program's execution, such that the program can be represented as a logical formula and explored systematically. Symbolic execution can then find concrete input values that execute a path by solving the corresponding path constraint using a SAT/SMT solver.

Although symbolic execution is a powerful automated analysis technique, it suffers from scalability problems and requires further improvement for practical use. One critical challenge is path explosion. When symbolic execution explores paths, it keeps track of all possible execution paths. Since the number of possible execution paths grows exponentially with branches, symbolic execution often consumes excessive computing resources after checking only a small portion of code. Path explosion gets worse when symbolic execution attempts to analyze shared libraries, which tend to be too complex for it to explore. To mitigate path explosion and improve code coverage, a line of research studies path exploration heuristics (such as breadth-first search, depth-first search and concolic) [20] so as to maximize code coverage within the given amount of time and computing resources. Another line of work studies *path pruning*, a technique that reduces search space by removing uninteresting paths before further exploration [19] [24]. Since path exploration heuristics are relatively well-studied, this work will focus on path pruning, particularly, pruning unsatisfiable paths.

Existing symbolic execution tools have implemented several path pruning strategies. Because all of the derived paths of an unsatisfiable path will remain unsatisfiable, at first thought it seems that symbolic execution could conserve resources by checking paths frequently and removing unsatisfiable paths as early as possible. On the other hand, checking a path's satisfiability takes non-negligible time; therefore, sometimes it might be more efficient to defer path checking and assume unchecked paths are currently satisfiable. It is unclear which strategy is optimal.

This paper formulates the **path pruning problem (PPP)**, demonstrates that static path pruning is a suboptimal solution to PPP, and proposes dynamic path pruning (DPP) to consistently achieve near-optimal exploration time. At a high level, the path pruning problem aims to minimize the execution time of symbolic execution by deciding whether and when to check a path's satisfiability. Static path pruning (i.e., each path is checked with a pre-defined probability regardless of the program) has unacceptable worst-case performance in solving PPP because the ratio of unsatisfiable paths differs from program to program. Moreover, the ratio also varies within a program and can also be influenced by the pruning process.

For this reason, we propose **dynamic path pruning (DPP)**, a path pruning strategy that consistently achieves near-optimal exploration time for a wide spectrum of programs. The intuition behind DPP is to assign a higher checking rate to paths that are more likely to be unsatisfiable; this likelihood is calculated based on observed information, such as the percentage of unsatisfiable paths among checked paths at each exploration layer. We model PPP as an optimization problem and derive the optimal checking rate. Specifically, we define the exploration penalty of four types of paths—(1) unsatisfiable and unchecked, (2) satisfiable and unchecked, (3) unsatisfiable and checked, and (4) satisfiable and checked—and try to minimize the total penalty during path exploration in symbolic execution. In addition, we also analyze the distribution of the time it takes to solve path constraints so as to

[1]A path is a sequence of basic blocks.

set a proper solver timeout that reduces the overall penalty.

As a proof of concept, we extend angr [1] [2] [3] [6], an open source symbolic execution platform, to support DPP. We compare DPP with different static checking rates using a diverse set of programs. The experiment results confirm that our dynamic path pruning strategy can significantly reduce exploration time and memory usage when compared with static path pruning strategies, whereas existing static path pruning strategies suffer from unacceptable worst-case performance due to their program-independent behaviors: Compared with static strategies, DPP performs best in 84% (110 out of 131) of CGC binaries, and the exploration time is within 100–124% of the best static strategy in 95% of the tested hand-crafted an coreutils binaries.

II. Background and Motivating Example

This paper proposes a dynamic path pruning technique to improve code coverage in symbolic execution. In this section, we provide relevant background information on symbolic execution and path pruning.

A. Symbolic Execution

Overview. Symbolic execution is an automated analysis technique that uses symbolic values to simulate program execution. In contrast to using concrete values to execute a program directly, symbolic execution systematically explores every execution path to a certain target (which can be the end of a program or a particular memory address).

In symbolic execution, inputs are represented as symbolic values, and an execution path is represented as a logical formula (i.e., first-order logic constraints over symbolic values) called *path predicate*. Since symbolic execution does not directly execute a program with concrete values, given a path predicate, we need to know if there is an assignment of concrete values to the symbolic variables such that the predicate is evaluated as true. If such an assignment exists, we say the path is *satisfiable*; otherwise, the path is *unsatisfiable*. Symbolic execution typically uses a SMT (satisfiability modulo theories) solver (e.g., Z3 [14]) to search for concrete values that satisfy the path predicate.

Path exploration. To explore possible paths, a symbolic executor maintains a worklist that contains 1) *satisfiable* and 2) *unchecked* paths. Unchecked paths are explored but have not checked for their satisfiability, and thus can be either satisfiable or unsatisfiable.

During exploration, the symbolic executor takes out a path from the worklist based on a certain exploration strategy (e.g., BFS or DFS). The path will be extended by one basic block to generate its successor(s). Specifically, when encountering a condition during exploration, symbolic executor branches it into two paths: one in which the condition is satisfied, and the other in which the condition is violated (i.e., the negated condition is satisfied). The path may branch into more paths if it encounters an indirect jump. For example, when encountering a condition $x > 10$, symbolic execution replaces the current path with two new paths, where one corresponds to $x > 10$ and the other corresponds to the negated condition, $x \leq 10$. These conditions and negated conditions are called path constraints (or constraints for short).

For each newly discovered path, the symbolic executor can choose to check its satisfiability immediately or skip the check. Unsatisfiable paths are discarded, and satisfiable or unchecked paths are added to the worklist.[2] At the end of path exploration, the symbolic executor validates all paths remaining in the worklist using a SMT solver.

B. Path Pruning in Symbolic Execution

Due to the path explosion problem, symbolic execution needs to allocate an excessive amount of memory to maintain the worklist. To reduce memory overhead, one common approach, which we refer to as *CheckAll*, is to immediately check the satisfiability of newly discovered path using the SMT solver, such that every unsatisfiable path is pruned as early as possible.

However, solving a path predicate is time-consuming, and the solver may timeout when encountering a complex predicate. The symbolic executor can sometimes gain by skipping a complex path predicate, as the subsequent paths spawn from it might be easier to solve. For example, angr supports the *CheckNothing* strategy, where all paths remain unchecked until the end of path exploration.

CheckAll and CheckNothing are two cmmon static path pruning strategies, whose decision of choosing to check a path is independent of the path predicate in the runtime. We observe that such static path pruning is suboptimal for the following two reasons:

- **Static path pruning cannot consistently minimize path exploration time for all programs.** While *CheckAll* performs better for programs with more unsatisfiable paths, *CheckNothing* performs better for programs with more satisfiable paths. However, without analyzing the program, it would be challenging to determine the percentage of unsatisfiable paths in each program. Moreover, neither *CheckAll* nor *CheckNothing* will perform well for a program that has many unsatisfiable paths at the beginning and many satisfiable paths with comlex predicates at the end.
- **The unsatisfiable path rate changes as symbolic execution proceeds.** It is difficult to accurately predict the percentage of unsatisfiable paths in the worklist without solving them. Moreover, because symbolic execution removes known unsatisfiable paths during exploration, the unsatisfiable path rate in the worklist will change over time and become even harder to predict.

III. Proposed Solution: Dynamic Path Pruning

This section presents dynamic path pruning (DPP), a path pruning strategy that automatically adjusts the checking rate according to a program's characteristics in order to minimize the expected exploration time and thus improve code coverage. We formulate the dynamic path prunning problem as an

[2] A path is considered unchecked if the solver times out.

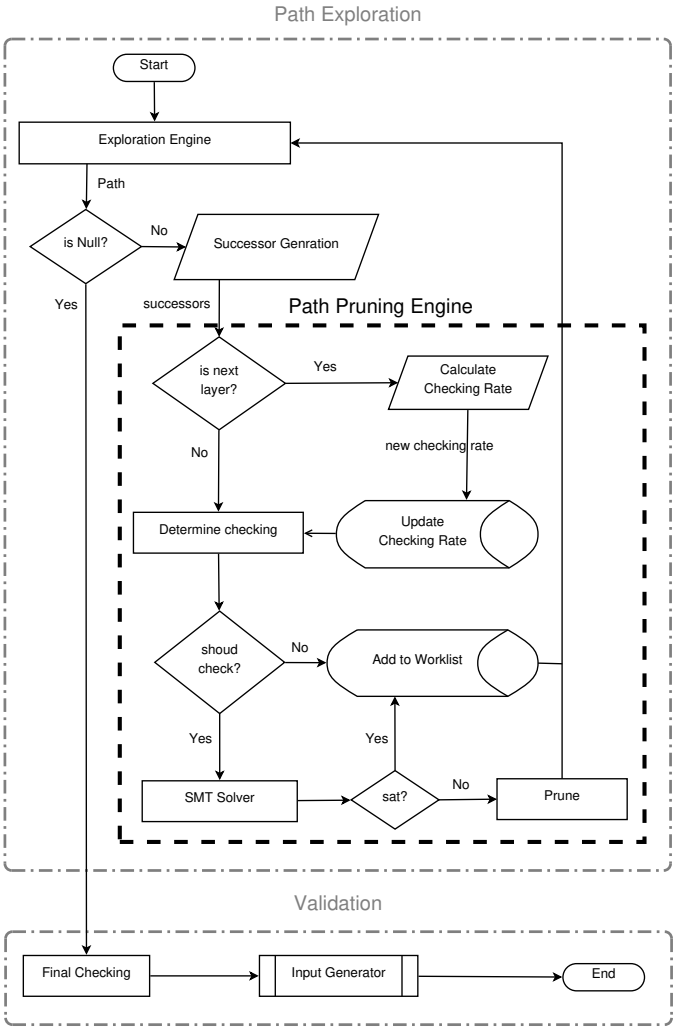


Fig. 1: Data Flow Diagram of Path Pruning

optimization problem, and find an optimal *checking rate* such that overall exploration time is minimized.

We first describe the high-level interface of DPP, and then describe the optimization objective and solution.

A. Path Pruning Interface

Figure 1 shows the location of the path pruning component in symbolic execution. When symbolic execution explores paths, the path pruning component is responsible of deciding whether to check a path at each step. The explorer uses a worklist to store paths and uses an exploration strategy which can be BFS, DFS, etc. When a path is explored to the next step (i.e., extended by one basic block), it will be passed to the path pruning engine.

If the path pruning engine decides to check a path according to its strategy, the path will be passed to a SMT solver. If the result of the SMT solver is satisfiable (SAT) or unknown (UNKNOWN), the path pruning engine will return *True* to the explorer. If the path is unsatisfiable, it will return *False* to the explorer. The path pruning engine directly returns *True* to the explorer if it decides not to pass a path to the SMT solver. The explorer adds the path to the worklist if it receives True from the path pruning engine; otherwise, the explorer prunes the path. After the explorer finishes exploring every path or reaches the target addresses, all of these remaining paths in the worklist will be checked by the SMT solver. In the final check, if the path is satisfiable, the SMT solver can generate an input that may trigger the execution of the path. Otherwise, the unsatisfiable or unknown paths will be pruned because the SMT solver cannot generate any input that will trigger the execution of the path.

B. DPP Overview

Given an explored path, DPP aims to compute a probability of checking the path so that the expected total exploration time is minimized. This probability is called the checking rate of the path. DPP determines this checking rate based on the observation of previously checked paths, and the checking rate can be dynamically adjusted. More specifically, the rate can be adjusted per binary, per layer, and per path.

This work focuses on per-layer optimization, which is nicely coupled with the BFS path exploration strategy. We observe that the unsatisfiable path rate of one layer is related to the unsatisfiable path rate of the previous layer. Based on the observed unsatisfiable path rate of one layer, we calculate the optimal checking rate for the next layer such that exploration time is minimized.

We acknowledge that per-path checking rate optimization may offer better performance than per-layer adjustment, because per-path is a generalized case of the other. We leave it as future work to study such finer-grained optimization.

C. DPP Pseudocode

Algorithm 1 presents our dynamic path pruning strategy.

Algorithm 1: Dynamic path pruning

```
   Input: each path which is explored
1  if next depth then
2      upr = estimate_unsat_path_rate(totalResults);
3      checkingRate = generate_new_checking_rate(upr);
4      clear totalResults;
5  if should_check(checkingRate) then
6      result = SMT_solve(path);
7      add result to totalResults;
8      if result is unsat then
9          prune(path);
```

In Lines 2 and 3, we can see that the path pruning engine calculates a new checking rate for this layer by using the previously observed unsatisfiable path rate. Then the path pruning engine applies the new checking rate for every path in this layer. For a given path, the `should_check` in Line 5 will determine whether the path should be checked uniformly at random based on the current per-layer checking

rate. If the path should be checked, the path will be passed to `SMT_solve` in Line 6 and be pruned if it is unsatisfiable.

D. Estimate Unsatisfiable Path Rate

Because the checking rate is not always 100%, some paths on this layer are left unchecked. After all the paths of this layer are explored, some unsatisfiable paths remain in the worklist. To estimate the ratio of unsatisfiable paths in the remaining paths, we can extrapolate based on the result of the checked paths (i.e., the ratio of unsatisfiable paths in the checked paths), and predict the unsatisfiable path rate on the next layer. Once we obtain an estimated unsatisfiable path rate for the next layer, we calculate the optimal checking rate that will minimize the exploration time of that next layer.

Because the checking rate is not always 100%, the total number of checked paths is always fewer than the total number of paths on this layer. Therefore, we use the ratio of observed unsatisfiable paths to estimate the unsatisfiable path rate on the next layer after checking all the paths on this layer:

$$U_{k+1} = (\frac{\hat{U_k}}{N_k \hat{C_k}} N_k(1-\hat{C_k}))\theta_k \quad (1)$$

$$S_{k+1} = (\frac{\hat{S_k}}{N_k \hat{C_k}} N_k(1-\hat{C_k}) + \hat{S_k})\theta_k \quad (2)$$

$\hat{U_k}$ represents the total number of observed unsatisfiable paths on a layer k, and $\hat{S_k}$ represents the total number of observed satisfiable paths on a layer k. To estimate the total number of unsatisfiable paths on k after checking all paths on this layer, the total number of estimated unsatisfiable paths is the observed unsatisfiable path rate $\frac{\hat{U_k}}{n_k \hat{C_k}}$ times the total number of unobserved paths $N_k(1-\hat{C_k})$, where N_k is the total number of paths on this layer and $\hat{C_k}$ is the checking rate. Similar to the estimated of total number of unsatisfiable paths, the total number of estimated satisfiable paths is the observed satisfiable path rate $\frac{\hat{S_k}}{n_k \hat{C_k}}$ times the total number of unobserved paths $N_k(1-\hat{C_k})$. However, the number of estimated satisfiable paths should include the number of observed satisfiable paths $\hat{S_k}$ because the observed satisfiable paths are not pruned. Finally, to estimate the total number of unsatisfiable paths on the next layer, we calculate the number of estimated unsatisfiable paths times the growing degree θ_k, which is obtained by averaging the observed degree of every checked paths. Note that we assume the growing degree of the next layer is the same in this layer, and both the satisfiable and unsatisfiable path rates are the same in two consecutive layers.

Bias & Confidence Level There are two potential biases in our estimation:

1) θ_k is estimated by averaging the observed paths rather than all paths. Also, the growing degrees of satisfiable and unsatisfiable paths might differ, because satisfiable paths can be extended into unsatisfiable paths.
2) If the checking rate is too low, we lack a sufficient confidence level to estimate the unsatisfiable path rate on this layer.

The first bias is unavoidable since we are unaware how many satisfiable paths will later change into unsatisfiable paths.

The second bias comes from the fact that a lower confidence level on one layer will result in a larger bias in the estimated number of unsatisfiable paths on the next layer. We argue that although such a bias may happen, its effect will not last indefinitely; we will be able to restore the confidence level after several layers. Given a low confidence level, there are two possibilities. First, the unsatisfiable path rate on the next layer is low but the estimated unsatisfiable path rate is high. In this case, the checking rate will be high for the next layer, and thus the confidence level. The second possibility is that the unsatisfiable path rate on the next layer is high but the estimated unsatisfiable path rate is low. In this case, the checking rate for the next layer will be low, and so does the confidence level. However, because the probability that this case occurs for consecutive layers decreases exponentially, the confidence level will be restored after a few layers. Moreover, we can set a lower bound on the unsatisfiable path rate to avoid a low confidence level. The (upper and lower) bounds of the unsatisfiable path rate are parameters that can be tuned in our system.

E. Minimizing Penalty

As mentioned before, DPP is an optimization problem that aims to minimize path exploration time. Here we formulate the optimization objective by defining the *penalty*—the additional time taken by path exploration compared to the ideal scenario.

What would be the ideal scenario? To simplify the problem, we assume the SMT solver can check every path at a constant rate of time and that the SMT solving time dominates the time of path exploration. We leave it to future work to optimize the solver and take other processing times into account.

Thus, the minimum checking cost at layer $k+1$ is $U_{k+1}E_c$, where U_{k+1} is the number of unsatisfiable paths on layer $k+1$ and E_c is the cost of checking a path. E_c represents the average solving time per path by using the SMT solver, and E_u represents the cost of removing all unsatisfiable paths that are derived from an unpruned unsatisfiable path on layer k. The minimum checking cost is $U_{k+1}E_c$.

Given this ideal scenario, we can define the penalty on layer k as the cost to check the sampled paths plus the implicit cost of unpruned unsatisfiable paths (that is, the expected cost of pruning all the derived unsatisfiable paths in the future) minus the ideal checking cost. Formally,

$$Pred(\hat{U_k}, \hat{S_k}) = C_{k+1}$$

find C_{k+1} s.t. the penalty in layer k (pe_k) is minimized:

$$\begin{aligned} pe_k =& C_{k+1}(U_{k+1} + S_{k+1})E_c \\ &+ (1-C_{k+1})U_{k+1}E_u - U_{k+1}E_c \end{aligned} \quad (3)$$

In this calculation, we want to evaluate the checking rate on the next layer by using the total number of observed unsatisfiable and satisfiable paths on this layer. To minimize the penalty, we should find a checking rate on the next layer

TABLE I: Binary Unsatisfiable Path Rate

Binary (Depth)	Unsatisfiable Path Rate (%)
binary1 (80)	0
binary2 (50)	99.88
binary3 (100)	95.93
grep (70)	54.05
readelf (65)	1.30
mkdir (75)	0
echo (65)	85.72
ls (75)	30.80
touch (65)	4.83
cp (90)	12.5

C_{k+1} that minimizes the total checking cost on the next layer $C_{k+1}(U_{k+1} + S_{k+1})E_c$ and the implicit cost of unpruned unsatisfiable paths $(1 - C_{k+1})U_{k+1}E_u$.

The relationship between E_u and E_c is

$$\begin{aligned} E_u =& C_{k+1}E_c \sum_{i=1}^{\infty} \theta^i (1 - C_{k+1})^{i-1} \\ =& \frac{C_{k+1}E_c}{1 - \theta(1 - C_{k+1})} \end{aligned} \tag{4}$$

To expand E_u (the cost) when a path grows into the next layer by a degree θ then some of them are pruned by using the checking rate C_{k+1}, is $\theta C_{k+1}E_c$. The cost, which the remaining unsatisfiable paths on the next layer $\theta(1-C_{k+1})$ are pruned by using the same checking rate C_{k+1}, is $\theta^2(1 - C_{k+1})C_{k+1}E_c$, and so on.

Finally, to find the optimal C_{k+1}, we differentiate pe_k with respect to C_{k+1}, and find C_{k+1} such that the differentiation of the penalty equals 0. Note that to ensure pe_k converges and is differentiable, $\theta(1 - C_{k+1}) < 1$ should hold.

IV. Evaluation

Test cases. To conduct a comprehensive evaluation, we use three sets of test cases in the performance comparison:

1) Three hand-crafted binaries: These three binaries are designed to test extreme cases. *binary1* has many unsatisfiable paths, *binary2* has no unsatisfiable paths but every path contains a large set of constraints (and the SMT solver will spend more than 0.01 seconds solving each of them), and *binary3* has many unsatisfiable paths at the beginning and generates a large number of constraints for each path at the end of the binary.
2) Seven coreutils: Coreutils are used to test our dynamic path pruning strategy in general cases. These seven binaries are `grep`, `readelf`, `mkdir`, `echo`, `ls`, `touch`, and `cp`.
3) 131 CGC binaries: For large-scale evaluation, we use the 131 binaries released from the DARPA Cyber Grand Challenge (CGC).

Before starting the experiments, we use *CheckNothing* to run every program for a fixed amount of time and obtain the unsatisfiable path rate. As Table I shows, different programs have different unsatisfiable path rates and each program has a different unsatisfiable path rate on a different layer.

Experiment setup & implementation. Our experiments run in a Linux environment, using an AMD64 processor E5-2620 and 64 GB memory. To ensure fair comparison, each test uses one thread on one server. We use angr, an open source binary analysis tool, as our symbolic execution engine. Note that angr's default setting uses *CheckAll*, and can switch to *CheckNothing* by setting the *LAZY_SOLVES* option. In order to perform a fair comparison among various strategies, we implement dynamic path pruning as a class that will be called whenever a new path is explored.

Evaluation metrics. We consider two evaluation metrics: coverage and memory usage. *Code coverage* is defined as the total number of explored basic blocks over time. A strategy with higher code coverage is better. However, code coverage does not capture the progress made within a loop because a strategy which explores the loop one time and a strategy which explores the loop 100 times cover the same number of basic blocks. Hence, we propose using *depth coverage* as an alternative measure when applying BFS for exploration. Specifically, we instruct symbolic execution to stop exploring when it reaches a certain depth, and compare the time each strategy takes to reach the same depth, with shorter time being better. Another metric is memory usage, which evaluates memory usage over time. To compare across various programs, we also define the *performance ratio* to be

$$\frac{\text{resource consumption}}{\text{minimum resource consumption of strategies}} \times 100\%$$

, which captures the closeness to optimality.

The default timeout value of the SMT solver is 0.3 seconds and we will analyze different timeout values in Section IV-C.

A. Depth Coverage

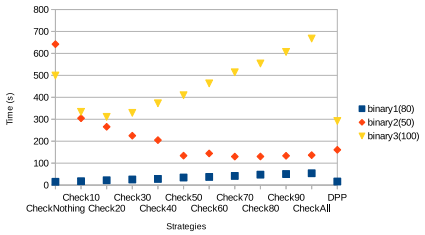
In this experiment, DPP's checking rate is empirically bounded within 5 ˜90% to ensure timely adjustment. *CheckX* is a static path pruning strategy with an X% checking rate. *CheckAll* and *CheckNothing* have 100% and 0% checking rates, respectively.

1) Three hand-crafted binaries: Figure 2a shows the time consumption of different strategies with three programs. Figure 2b shows the performance ratio of different strategies with three binaries.

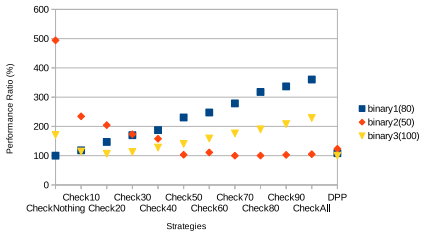
In this figure, *CheckNothing* is better than *CheckAll* in the first binary because the first binary has a higher unsatisfiable path rate. However, *CheckAll* is better than *CheckNothing* in the second binary because it has a 99% unsatisfiable path rate. *DPP* is not the best strategy in the first and second binaries, but it comes close to the execution time of the best strategy.

In the third binary, *DPP* is the best strategy, while *CheckNothing* and *CheckAll* suffer from high exploration time.

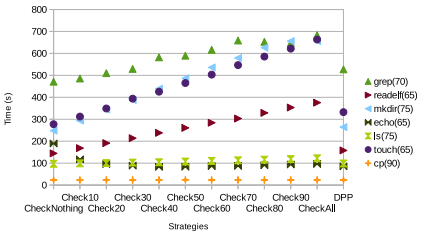
Recall that the unsatisfiable path rate of binary2 is 99.87 %, as shown in Table I, but the best static strategy for binary2 is *Check70* rather than *Check100* (the closest rate to the unsatisfiable path rate). The reason for this is that, as we prune the unsatisfiable paths, the unsatisfiable path rate of the remaining paths will change. Thus, the best strategy among all the static strategies may change over time. Dynamic path



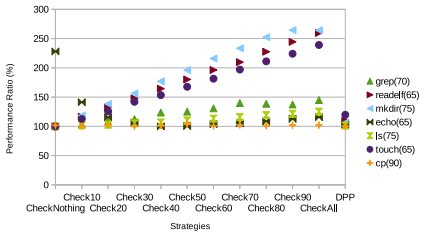
(a) Depth Coverage of Three Special Binaries



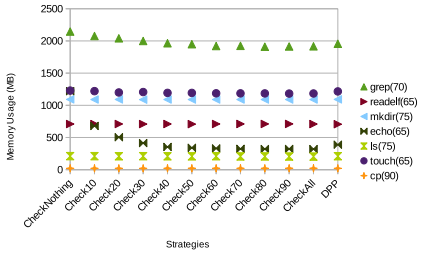
(b) Normalized Time of Depth Coverage of Three Special Binaries



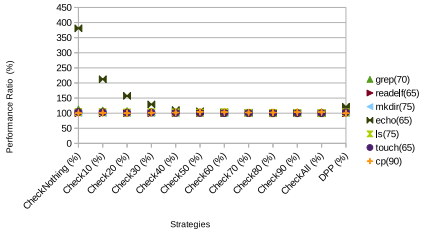
(c) Depth Coverage of Coreutils



(d) Normalized Time of Depth Coverage of Coreutils



(e) Memory Usage of Coreutils



(f) Normalized Memory Usage of Coreutils

Fig. 2: Experiment results

pruning avoids this problem because the checking rate can be adjusted dynamically based on the current unsatisfiable path rate on every layer.

2) Coreutils: Figure 2c shows the time consumption of different strategies with coreutils. Figure 2d shows the performance ratio of different strategies with coreutils.

The results confirm that DPP can ensure low exploration time in a diverse set of programs and can avoid the worst-case exploration time.

Table II summarizes the normalized time for each strategy. We can see that although *Check0* (*CheckNothing*) is the best for many programs, *Check0* occasionally performs the worst (e.g., for *binary2* and *echo*). *Check100* (*CheckAll*) also performs the worst for many programs. On the other hand, *DPP* always approximates the best time and does not have any worst-case performance. As for the average and standard deviation, we can see that *DPP*'s average is 108.43 and the standard deviation is 8.13, which means that when we analyze a new program, there is a 95% probability that the exploration time will be within 100–124.69% compared to the best strategy.

3) CGC Binaries: Figure 3 shows the performance ratio of three path pruning strategies (DPP, CheckAll, and CheckNothing) over CGC binaries. DPP outperforms CheckAll and CheckNothing in 110 out of the 131 CGC binaries. Although DPP is 5-8x slower in 23 out of 133 binaries (e.g., CROMU_00004), manual inspection reveals that such binaries have uncommon characteristics (e.g., having no branch at all), such that DPP's computational overhead dominates the overall cost. We can address these uncommon cases by avoiding frequent update of the checking rate.

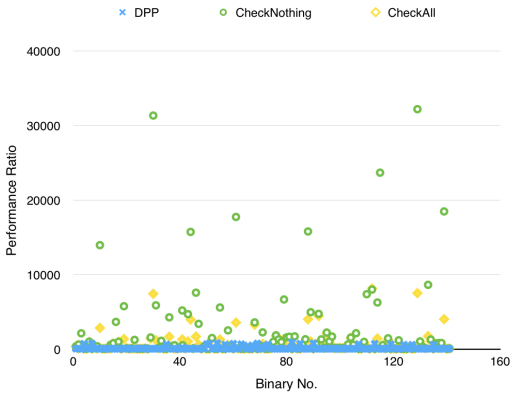


Fig. 3: CGC binaries

B. Memory Usage

Figures 2e, and 2f are the results of a memory usage experiment with same configuration of depth coverage. Table III shows all the memory usage of each strategy. Table II shows that *CheckNothing* is the most efficient for many programs. However, for some programs such as *echo*, *CheckNothing* uses twice as much memory than the other strategies. This is because *CheckNothing* kept too many unsatisfiable paths. On the other hand, *DPP*'s memory usage is slightly higher than *CheckAll*, but it is more efficient than *CheckAll* as shown previously.

C. Timeout

To evaluate the impact of solver timeout value, we compare the time an SMT solver, Z3, takes to solve a path, and compare the total execution time when symbolic execution uses a different timeout.

TABLE II: Normalized Time of Depth Coverage

Binary (Depth)	Best Time (s)	Normalized Time (%)											
		Static Path Pruning Strategies											DPP
		0	10	20	30	40	50	60	70	80	90	100	
binary1 (80)	14.97	100	118	147	170	187	231	248	279	318	337	360	109
binary2 (50)	129.93	494	234	204	173	158	103	111	100	100	103	105	124
binary3 (100)	291.96	171	114	107	113	128	140	159	176	190	208	229	100
grep (70)	471.03	100	103	108	112	124	125	131	140	139	137	145	112
readelf (65)	144.65	100	116	132	148	164	180	196	210	228	244	260	109
mkdir (75)	248.20	100	118	138	156	177	196	216	233	252	264	264	106
echo (65)	83.09	228	141	116	107	100	101	104	105	109	113	116	103
ls (75)	97.36	100	102	103	105	107	110	113	116	119	122	125	101
touch (65)	277.44	100	113	126	142	153	168	181	197	211	224	239	120
cp (90)	22.93	102	100	102	100	101	104	101	103	101	102	102	100
average percent		159.49	126.08	128.35	132.60	139.90	145.76	156.01	165.84	176.62	185.39	194.53	108.43
standard deviation		125.36	39.88	30.93	28.31	32.21	45.61	52.19	62.80	74.89	81.70	88.03	8.13

TABLE III: Normalized Memory Usage

Binary (Depth)	Best Memory Usage (MB)	Normalized Memory Usage (%)											
		Static Path Pruning Strategies											DPP
		0	10	20	30	40	50	60	70	80	90	100	
binary1 (80)	14.94	103	100	105	103	103	103	100	105	103	103	103	106
binary2 (50)	37.38	5826	1301	400	274	191	146	127	112	113	104	100	252
binary3 (100)	142.70	281	224	182	153	125	119	108	103	100	101	102	165
grep (70)	1910.28	112	109	107	105	103	102	101	101	100	100	100	102
readelf (65)	706.67	100	100	100	100	100	100	100	101	101	101	100	100
mkdir (75)	1088.21	100	100	100	100	100	100	100	100	100	100	100	100
echo (65)	319.97	381	212	157	129	110	106	103	101	100	101	100	121
ls (75)	203.27	103	103	102	103	102	102	103	101	101	101	100	103
touch (65)	1182.31	104	103	101	102	101	101	100	100	100	100	100	103
cp (90)	20.51	100	101	100	101	102	104	104	102	102	104	104	100
average percent		721.14	245.40	145.57	126.96	113.75	108.32	104.64	102.60	102.05	101.37	100.93	125.25
standard deviation		1,796.15	373.97	93.76	54.32	28.04	14.33	8.25	3.50	4.07	1.59	1.35	48.81

An ideal timeout should be long enough to find most unsatisfiable paths, while short enough to avoid checking satisfiable paths. If the timeout is too long, few complex queries can dominate the overall exploration time. If the timeout is too short, the SMT solver may timeout and return UNKNOWN before being able to draw any conclusion, and thus many unsatisfiable paths will be left unpruned. In other words, checking time is wasted.

Our experiment shows that, for many programs and layer depth, 80% of unsatisfiable paths can be identified in less than 0.01 seconds. Also, the majority of satisfiable paths take longer to be solved. Hence, we set the E_c value (discussed in Section III) to 0.01–0.02 seconds, sufficient for checking and pruning unsatisfiable paths.

Z3 can solve unsatisfiable paths in 0.01 seconds for almost every program, but it takes between 0.01–1 second for the solver to solve a satisfiable path. In Figures 4a, 4b, 4c, and 4d, we can see that by setting the timeout to 0.02 seconds, we can solve most of the unsatisfiable paths without wasting too much time on satisfiable paths.

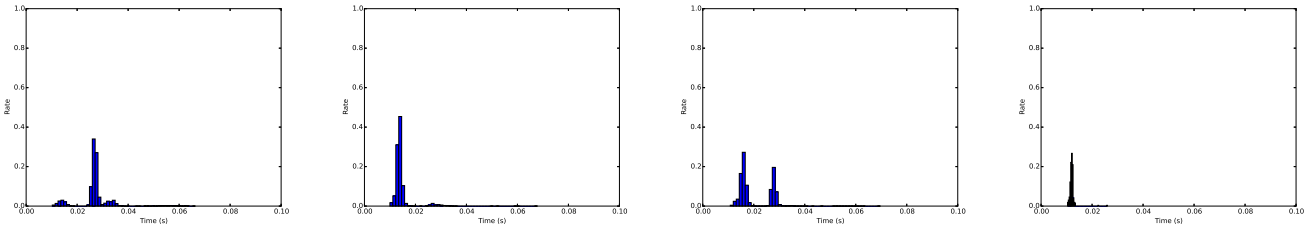
We use *CheckAll* with different timeouts to analyze total execution time. The timeouts are 20, 300, and 10000 milliseconds. A long timeout allows symbolic execution to find more unsatisfiable paths, but will also increase the execution time.

V. Related Work

Reducing program state. Path explosion is one of the main obstacles to efficient symbolic exeuction. Some research proposes techniques to reduce the amount of symbolic state. Veritesting [7] is a technique that combines static symbolic execution (SSE) and dynamic symbolic execution (DSE). It uses DSE to explore paths and SSE to merge path predicates in order to mitigate path explosion. In DASE [24], the uninteresting path is pruned by adding constraints to the input according to documents. Pre-added constraints allow symbolic execution to focus on exploring interesting paths. Another technique regarding uninteresting paths is the under-constrained symoblic exeuction [19]. Under-constrained symbolic execution compares and identifies different paths between patched and non-patched binaries so as to find new bugs in the patched binary, rather than the non-patched one. Another technique is heuristic path pruning [11], which tries to find non-fatal error handling branch (NFEHB) patterns and only forks a path which is not NFEHB, because they observe that few vulnerabilities are in non-fatal error handling in Common Vulnerabilities and Exposures (CVE). In other words, it tries to prune paths that are NFEHB. Our work can be easily integrated with these techniques.

Automated analysis. Manual code or binary analysis is time consuming. To improve performance, automated techniques such as symoblic execution and fuzzing [25] have received increasing attention lately. Symbolic execution was introduced in 1975 [17] [18] and have decades of history [10]. DART [15] proposes concolic testing to significantly improve the efficiency of symbolic execution.

Open source tools. Several open-source symbolic execution tools are widely used in practice. Cristan et al. developed EXE [9], an effective bug-finding symbolic executor. It features modeling of memory and optimization for constraint solving. Inheriting advantages from EXE, KLEE [5] proves



(a) Satisfiable path in grep (b) Unsatisfiable path in grep (c) Satisfiable path in readelf (d) Unsatisfiable path in readelf

Fig. 4: Distribution of per-path solving time

that symbolic execution can find bugs in real-world programs. To scale to large systems, Chipounov et al. presented selective symbolic execution (S2E) [12], [13]. S2E can reduce the amount of code being executed symbolically, and can combine with virtualization. TRITON and angr are dynamic binary analysis tools. Although binary based tools lose some information, they can use dynamic techniques like CFG recovery or backward slicing [22] to obtain binary information. The differences between TRITON and angr are that TRITON uses pin [4] (which can analyze x86 and x86-64 instructions set) to trace the execution of a binary. angr, on the other hand, can analyze any binary on any platform. There are some extensions of angr such as firmalice [21] and Driller [23]. Another notable symoblic execution tool is SAGE [16], which found critical bugs during the development of Windows 7 [8].

VI. Conclusion

Automated program analysis has attracted significant attention over the past decades as software applications advance in scale and complexity. Symbolic execution is a powerful automated analysis technique that systematically explores a program to find crashing inputs and reproduce bugs. However, because the number of possible execution paths grows exponentially with branches, symbolic execution suffers from path explosion and often fails to find bugs further away from the entry point.

This paper investigates the path pruning problem, which aims to minimize path exploration time in symbolic execution by deciding whether and when to check a path's satisfiability. To overcome the limitations of static path pruning strategies, we propose dynamic path pruning (DPP), in which the checking rate is dynamically adjusted based on the program's characteristics. Our evaluation shows that DPP's exploration time and memory usage are close to the best static path pruning strategies in a diverse set of programs.

References

[1] "angr," http://angr.io/.
[2] "Angry hacking," DEFCON 2015.
[3] "A dozen years of shellphish from defcon to the cyber grand challenge," HITCON 2015.
[4] "Pin - a dynamic binary instrumentation tool," https://software.intel.com/en-us/articles/pintool.
[5] "KLEE LLVM execution engine," http://klee.github.io/.
[6] "Using static binary analysis to find vulnerabilities and backdoors in firmware," BLACK HAT 2015.
[7] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *Proceedings of the 36th International Conference on Software Engineering*, 2014.
[8] E. Bounimova, P. Godefroid, and D. Molnar, "Billions and billions of constraints: Whitebox fuzz testing in production," in *Proceedings of International Conference on Software Engineering*, 2013.
[9] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically Generating Inputs of Death," in *ACM CCS*, 2006.
[10] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," in *Communications of the ACM*, vol. 56, no. 2, 2013, pp. 82–90.
[11] D. Chen, Y. Zhang, L. Cheng, Y. Deng, and X. Sun, "Heuristic path pruning algorithm based on error handling pattern recognition in detecting vulnerability," in *IEEE Computer Software and Applications Conference Workshops (COMPSACW)*, 2013.
[12] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea, "Selective Symbolic Execution," in *Proceedings of Workshop on Hot Topics in System Dependability (HotDep)*, 2009.
[13] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems," in *ACM ASPLOS*, 2011.
[14] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.
[15] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *ACM SIGPLAN Notices - Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, vol. 40, no. 6, 2013, pp. 213–223.
[16] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, p. 20, 2012.
[17] W. Howden, "Methodology for the generation of program test data," in *IEEE Transactions on Computers*, vol. 24, no. 5, 1975, pp. 554–560.
[18] J. C. King, "Symbolic execution and program testing," in *Communications of the ACM*, vol. 19, no. 7, 1976, pp. 385–394.
[19] D. A. Ramos and D. Engler, "Under-constrained symbolic execution: correctness checking for real code," in *USENIX Security Symposium*, 2015.
[20] K. Sen, "Concolic testing," in *Proceedings of IEEE/ACM international conference on Automated software engineering*, 2007.
[21] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware," in *NDSS*, 2015.
[22] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *IEEE Symposium on Security and Privacy*, 2016.
[23] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *NDSS*, 2016.
[24] E. Wong, L. Zhang, S. Wang, T. Liu, and L. Tan, "DASE: Document-assisted symbolic execution for improving automated software testing," in *Proceedings of IEEE/ACM International Conference on Software Engineering*, 2015.
[25] M. Zalewski, "American fuzzy lop," http://lcamtuf.coredump.cx/afl/.