Published in IEEE Transactions on Dependable and Secure Computing, 2022, 19, 1955-1973. DOI: 10.1109/TDSC.2020.3042259

Probabilistic Path Prioritization for Hybrid Fuzzing

Lei Zhao, Pengcheng Cao, Yue Duan, Heng Yin, and Jifeng Xuan, *Member, IEEE*

Abstract—Hybrid fuzzing that combines fuzzing and concolic execution has become an advanced technique for software vulnerability detection. Based on the observation that fuzzing and concolic execution are complementary in nature, state-of-the-art hybrid fuzzing systems deploy "optimal concolic testing" and "demand launch" strategies. Although these ideas sound intriguing, we point out several fundamental limitations in them, due to unrealistic or oversimplified assumptions. Further, we propose a novel "discriminative dispatch" strategy and design a probabilistic hybrid fuzzing system to better utilize the capability of concolic execution. Specifically, we design a Monte Carlo-based probabilistic path prioritization model to quantify each path's difficulty, and then prioritize them for concolic execution. Our model assigns the most difficult paths to concolic execution. We implement a prototype named DigFuzz and evaluate our system with two representative datasets and real-world programs. Results show that the concolic execution in DigFuzz outperforms than those in state-of-the-art hybrid fuzzing systems in every major aspect. In particular, the concolic execution in DigFuzz contributes to discovering more vulnerabilities (12 versus 5) and producing more code coverage (18.9 versus 3.8 percent) on the CQE dataset than the concolic execution in Driller.

Index Terms—Software security, fuzz testing, concolic execution, hybrid fuzzing

1 Introduction

Software vulnerability is considered one of the most serious threats to cyberspace. Thus, it is crucial to discover vulnerabilities in a piece of software [11], [18], [24], [26], [31]. Recently, hybrid fuzzing, a combination of fuzzing and concolic execution, has become increasingly popular in vulnerability discovery [5], [12], [27], [30], [38], [41], [46]. Since fuzzing and concolic execution are complementary in nature, combining them can potentially leverage their unique strengths as well as mitigate weaknesses. More specifically, fuzzing is proficient in exploring paths containing *general* branches (branches that have large satisfying value spaces), but is by design incapable of handling paths containing *specific* branches (branches that have very narrow satisfying value spaces) [26]. In contrast, concolic execution can quickly generate concrete inputs that ensure the program to execute along a specific execution path, but it suffers from the path explosion problem [8]. In a hybrid scheme, fuzzing normally undertakes the majority tasks of path exploration due to the high throughput, and concolic execution assists fuzzing in exploring paths with low probabilities and generating inputs that satisfy specific branches. In this way, the path explosion problem in concolic execution is alleviated, as concolic execution is only responsible for exploring paths with low probabilities that may block fuzzing.

The key research question is how to combine fuzzing and concolic execution to achieve the best overall performance. Driller [38] and hybrid concolic testing [27] take a "demand launch" strategy: fuzzing starts first and concolic execution is launched only when the fuzzer cannot make any progress for a certain period of time, a.k.a., stuck. A recent work [41] proposes "optimal concolic testing". It quantifies the costs for exploring each path by fuzzing and concolic execution respectively and chooses the more economic method.

We have evaluated both strategies using the DARPA CQE dataset [15] and 12 real-world programs, and find that although these strategies sound intriguing, none of them work adequately, due to unrealistic or oversimplified assumptions.

For the "demand launch" strategy, first of all, the stuck state of a fuzzer is not a good indicator for launching concolic execution. Fuzzing is making progress does not necessarily mean concolic execution is not needed. A fuzzer can still explore new code, even though it has already been blocked by many specific branches while the concolic executor is forced to be idle simply because the fuzzer has not been blocked yet. Second, this strategy does not recognize specific paths that block fuzzing. Once the fuzzer gets stuck, the "demand launch" strategy feeds all seeds retained by the fuzzer to concolic execution for exploring all missed paths. Concolic execution is so overwhelmed by this massive number of missed paths that it may not generate a helping input for a specific path for a long time. By then, the fuzzer might have already generated a good input to

- *Lei Zhao and Pengcheng Cao are with the School of Cyber Science and Engineering, Wuhan University, Wuhan, Hubei 430072, China. E-mail: {leizhao, caopengcheng}@whu.edu.cn.*
- *Yue Duan is with the Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616 USA. E-mail: yduan12@iit.edu.*
- *Heng Yin is with the Department of Computer Science and Engineering, University of California, Riverside, CA 92521 USA. E-mail: heng@cs.ucr.edu.*
- *Jifeng Xuan is with the School of Computer Science, Wuhan University, Wuhan, Hubei 430072, China. E-mail: jxuan@whu.edu.cn.*

traverse that specific path, rendering the whole concolic execution useless.

Likewise, although the "optimal concolic testing" aims to make optimal decisions, it is nontrivial to quantify the costs of fuzzing and concolic execution for exploring each path. For example, the "optimal concolic testing" only estimates the cost of constraint solving in concolic execution based on constraints complexities, which neglects the cost of symbolic emulation. A recent study [14] shows that symbolic emulation is even more expensive than constraint solving. Furthermore, the "optimal concolic testing" deploys Markov Decision Processes (MDPC for short) to quantify the cost of fuzzing. This algorithm itself is expensive. As a result, the overall throughput of MDPC is greatly reduced. Finally, even if the costs of fuzzing and concolic execution can be accurately estimated, it is challenging to normalize them for a unified comparison, because these two costs are estimated by techniques with different metrics.

Based on these observations, we argue for the following design principles when building a hybrid fuzzing system: 1) since concolic execution is several orders of magnitude slower than fuzzing, we should only let it solve the "hardest problems", and let fuzzing take the majority task of path exploration; and 2) since high throughput is crucial for fuzzing, any extra analysis must be lightweight to avoid adverse impact on the performance of fuzzing.

In this paper, we design a "discriminative dispatch" strategy and propose a probabilistic hybrid fuzzing system to better combine fuzzing and concolic execution. That is, we prioritize paths so that concolic execution *only* works on selected paths that are *most* difficult for fuzzing to breakthrough. Therefore, the capability of concolic execution is better utilized. The key to our strategy is a lightweight method that quantifies the difficulty level for each path. Prior work solves this problem by performing expensive symbolic execution [20], and thus is not suitable for us.

In particular, we propose a novel Monte Carlo based probabilistic path prioritization (MCP^3) model to quantify each path's difficulty in an efficient manner. Specifically, we quantify a path's difficulty by its probability of how likely a random input can traverse this path. To calculate, we use the Monte Carlo method [34]. The core idea is to treat fuzzing as a random sampling process, consider random executions as samples to the whole program space, and calculate each path's probability based on the sampling information.

We have implemented a prototype called DigFuzz. It leverages a popular fuzzer, American Fuzzy Lop (AFL) [47], as the fuzzing component, and builds the concolic executor on top of two symbolic engines, *angr* [37] and *QSYM* [46]. In our previous work [48], DigFuzz only works on the CQE binaries from DARPA Cyber Grand Challenge [15] and the performance of DigFuzz on real-world programs are not demonstrated, because the concolic executor (*angr* [37]) does not have sufficient support for analyzing large real-world programs. In this study, we extend our prototype system by leveraging a second concolic executor (*QSYM* [46]) to analyze real-world programs. Then, we evaluate the effectiveness using the CQE binaries from the DARPA Cyber Grand Challenge [15], the LAVA dataset [17], and 12 real-world programs. The evaluation results show that the concolic execution in DigFuzz contributes significantly more to the increased code coverage (18.9 versus 3.8 percent) and increased number of discovered vulnerabilities (12 versus 5) than state-of-the-art hybrid systems, such as Driller [38]. Besides, we show that DigFuzz produces more code coverage on all real-world programs.

The contributions of the paper are as follows:

- We conduct an independent evaluation of two state-of-the-art hybrid fuzzing strategies ("demand launch" and "optimal concolic testing"), and discover several important limitations that have not been reported before.
- We propose a novel "discriminative dispatch" strategy as a better way to construct a hybrid fuzzing system. It follows two design principles: 1) let fuzzing conduct the majority task of path exploration and only assign the most difficult paths to concolic execution; and 2) the quantification of path difficulties must be lightweight. To achieve these two principles, we design a Monte Carlo based probabilistic path prioritization model.
- We implement a prototype system DigFuzz, and evaluate its effectiveness using the DARPA CQE dataset, LAVA dataset, and real-world programs. Our experiments demonstrate that DigFuzz outperforms state-of-the-art hybrid systems with respect to more discovered vulnerabilities and higher code coverage.

We have made our prototype and the evaluation datasets publicly available at https://github.com/Cc-tec/digfuzz.

2 Background and Motivation

In this section, we present background knowledge of the state-of-the-art hybrid fuzzing systems and then summarize several limitations of their strategy to motivate our research.

Fuzzing [29] and concolic execution [8] are two representative techniques for software testing and vulnerability detection. With the observation that fuzzing [29] and concolic execution [8] can complement each other, a series of hybrid fuzzing techniques [5], [27], [30], [38], [41] have been proposed. In general, these systems fall into two categories: "optimal concolic testing" and "demand launch".

To assess the performance of "optimal concolic testing" and "demand launch", we evaluate these two strategies using a dataset of 118 binaries from the CQE dataset and 12 real-world programs. In detail, we deploy the American Fuzzy Lop (AFL) [47] as the fuzzing component, and utilize *angr* [37] as the concolic executor to evaluate CQE binaries because of their specific and customized syscalls. Besides, since *angr* does not have sufficient support for real-world programs [31], we utilize *QSYM* [46], a state-of-the-art open-source concolic executor, to evaluate 12 real-world programs. Every binary is analyzed for 12 hours. Based on these two evaluations, we can draw several observations.

2.1 Optimal Concolic Execution

The "optimal concolic testing" aims to make an optimal choice to decide which technique (fuzzing or concolic execution) should be deployed to explore a given path [41]. To

TABLE 1
Execution Time Comparison

	Fuzzing	MDPC	*angr**	*QSYM**
Minimum	0.0007s	0.16s	18s	15s
25% percentile	0.0013s	13s	767s	102s
Median	0.0019s	65s	1777s	218s
Average	0.0024s	149s	1790s	264s
75% percentile	0.0056s	213s	2769s	588s
Maximum	0.5000s	672s	3600s	3363s

**One concolic execution is limited for at most 1 hour.*

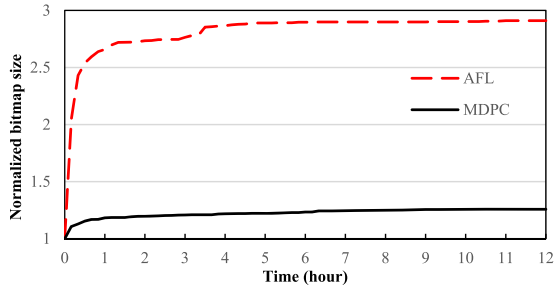


Fig. 1. Code coverage comparison between fuzzing and MDPC.

achieve optimal performance, it quantifies the cost of fuzzing and concolic execution, and always selects the method at a lower cost.

The key challenge is that the actual costs of fuzzing and concolic execution are extremely difficult to estimate. The "optimal concolic testing" proposes an approximation technique to estimate these two costs. Specifically, it estimates the probability of program paths based on Markov Decision Processes (MDPC for short) and estimates the cost of constraint solving in concolic execution based on experience.

Observations. Based on the evaluations on two datasets, we have several observations.

- *Heavyweight Estimation*. Table 1 shows the execution time comparison among fuzzing, concolic execution (*angr*), concolic execution (*QSYM*), and MDPC from "optimal concolic testing". We can observe that estimating probability using MDPC is very costly, thousands of times slower than fuzzing.
- *Inaccurate Estimation*. By regarding the cost for a random test as 1, the "optimal concolic testing" model assumes that solving linear equality has a cost of 4, solving non-linear equality has a cost of 10, and solving a boolean combination of non-linear equalities has a cost of 50. However, Table 1 indicates that the cost of concolic execution varies greatly from program to program (from 18s to 3600s). Besides, it varies greatly for different concolic execution engines (*angr* and *QSYM*). These results further suggest that the estimation of MDPC is inaccurate.
- *Reduced Throughput*. Making decisions before exploration for each path significantly reduces the overall analysis throughput, from 417 executions per second in pure fuzzing to 2.6 executions per second with MDPC. Due to the negative impact of the reduced throughput, "optimal concolic testing" discovers fewer vulnerabilities and code coverage. To be more specific, Fig. 1 shows the bitmap size that is maintained by AFL, which is an approximation of code coverage. We can observe that "optimal concolic testing" discovers less code coverage than pure fuzzing. Moreover, "optimal concolic testing" detects vulnerabilities only in 29 CQE binaries, whereas the pure fuzzing can discover vulnerabilities in 67 CQE binaries. Similarly, among the 12 Linux binaries, "optimal concolic testing" only detects 3 crashes, whereas the pure fuzzing can discover 107 crashes.

We further optimize the system to have fuzzing and optimal decision work in parallel instead of running sequentially as in the original system and build a concurrent MDPC. We then evaluate it using the same dataset and have the following observation.

- Nearly all the missed paths are decided to be explored by concolic execution in several seconds after the fuzzing starts. By examining the coverage statistics, we observe that the fuzzer is able to generate hundreds of test cases in seconds, which leads to a high cost for exploring a missed path by fuzzing, based on the algorithm in MDPC. On the contrary, the cost of concolic execution is smaller even we assign the highest solving cost (50 as defined [41]) to every path constraint. This result indicates that the cost of concolic execution is underestimated.

Limitations. The aforementioned observations uncover the key problem of the "optimal concolic testing" strategy. That is, accurately estimating the cost for exploring a given path by fuzzing and concolic execution is nontrivial, which overshadows the benefit of making optimal solutions.

First, the estimation itself is so expensive that it reduces the throughput of the whole system. Specifically, MDPC leverages a path-sensitive program analysis for estimating the cost of fuzzing, which can be unacceptably heavyweight for programs with large states.

Second, the estimation is inaccurate. The "optimal concolic testing" assumes the cost of constraint solving is the weighted sum of the primitive operations (e.g., the Add and Mul operation) in the constraint, they then estimate the weight of each primitive operation type. Actually, concolic execution contains two steps, symbolic emulation, and constraint solving. A recent study shows that symbolic emulation is even more expensive than constraint solving [14].

Third, it is exceptionally challenging to define a unified metric to evaluate the cost of different techniques as a result of the huge distinctions among these techniques. Our evaluation shows the cost of concolic execution varies greatly from program to program, rendering the estimation of the cost unreliable. Furthermore, the cost of concolic execution also depends heavily on specific concolic executors (*angr* versus *QSYM*). How to improve the performance of concolic executors is an orthogonal problem [14].

To sum up, the "optimal concolic testing" constructs a theoretical framework for improving the performance of hybrid fuzzing. The proposed estimation technique, however, is impractical due to the limitations described above.

2.2 Demand Launch

The state-of-the-art hybrid schemes such as Driller [38] and hybrid concolic testing [27] deploy a "demand launch"

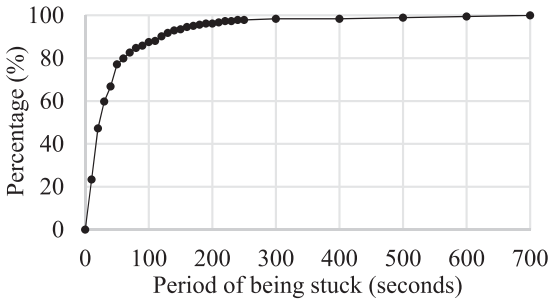


Fig. 2. The cumulative distribution of the stuck state duration.

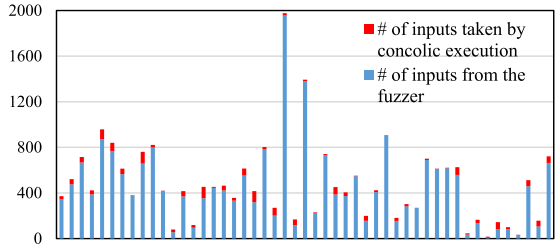


Fig. 3. The number of inputs retained by the fuzzer and the number of inputs processed by concolic execution.

strategy. In Driller [38], the concolic executor remains idle until the fuzzer cannot make any progress for a certain period of time. It then processes all the retained inputs from the fuzzer sequentially, to generate inputs that might help the fuzzer and further lead to new code coverage. Similarly, hybrid concolic testing [27] obtains both a deep and wide exploration of program state space via hybrid testing. It reaches program states quickly by leveraging the ability of random testing and then explores neighbor states exhaustively with concolic execution.

In a nutshell, two assumptions must hold in order to make the "demand launch" strategy work as expected:

1) A fuzzer in the non-stuck state means the concolic execution is not needed. The hybrid system should start concolic execution only when the fuzzer gets stuck.
2) A stuck state suggests the fuzzer cannot make any progress in discovering new code coverage in an acceptable time. Moreover, the concolic execution is able to find and solve the hard-to-solve condition checks that block the fuzzer so that the fuzzing could continue to discover new code coverage.

Observations. Based on the evaluation of how the "demand launch" strategy works on 118 binaries from the DARPA Cyber Grand Challenge (CGC) and 12 real-world programs, we find five interesting yet surprising observations.

- *A Low Percentage of Invoking Concolic Execution*. The "demand launch" strategy invoked concolic execution on only 49 out of the 118 CQE binaries, and 11 out of the 12 real-world programs. It means that the fuzzer got stuck on only these 60 binaries (49 CQE binaries and 11 real-world programs). This observation is on par with the numbers (42) reported in the paper of Driller [40].
- *Small Stuck Time Periods*. For the 60 binaries, on which concolic execution was invoked, we statistically calculated the stuck time periods, and the distribution of stuck time periods is shown in Fig. 2. We can observe that more than 85 percent of the stuck time periods were under 100 seconds.
- *Huge Throughput Gap*. The execution time in Table 1 shows that the throughput of fuzzing was orders of magnitude higher than the throughput of concolic execution. Therefore, a practical design should only pick very few inputs for concolic execution, as opposed to the "demand launch" strategy, which gives all the inputs retained by fuzzing to the concolic executor.
- *Low Practicability*. Due to such a huge throughput gap, in practice, the concolic execution is of little help to fuzzing in the "demand launch" strategy. Fig. 3 shows the number of inputs processed by concolic execution (*angr*) and the number of inputs retained by fuzzing (1709 out of 23915) for the CQE binaries. And for the 12 real-world programs, Table 2 shows that 795 inputs were processed by *QSYM*, out of a total of 15269 inputs retained by the fuzzer. On average, only 6.3 percent of the inputs retained by the fuzzer were processed by the concolic executor within the 12 hours of testing.
- *A Small Contribution to Code Coverage*. By the end of the 12 hours of testing, the "demand launch" strategy invoked concolic execution on 49 CQE binaries and finished 1709 runs. A more detailed investigation shows that, in total, only 51 inputs generated by concolic execution were imported by fuzzing, and the concolic execution was able to assist fuzzing on only 13 binaries, by generating at least one input that leads to new code coverage.

Limitations. The aforementioned results indicate two major limitations of the "demand launch" strategy.

First, the stuck state of a fuzzer is not a good indicator to decide whether the concolic execution is needed. According to the low percentage of invoking concolic execution, the fuzzer only gets stuck on 49 binaries, meaning concolic execution is never launched for the other 77 binaries. Manual investigation on the source code of these 77 binaries shows that they all contain specific branches that can block

TABLE 2
Inputs Retained by Fuzzer and pzrocessed by *QSYM*

	# of inputs by fuzzers	# of inputs by *QSYM*
libjpeg-9b+cjpeg	885	69
ncurses-6.1 +infotocap	2994	103
jhead-3.00	1109	46
nm-2.26.1	93	92
xpdf-4.00+pdfimages	1377	87
libpng-1.6.36+pngfix	938	34
readelf-2.26.1	4593	160
size-2.26.1	30	28
libtiff-4.0.10 +tiffdump	963	0
libxpat-R.2.2.5 +xmlwf	651	34
objdump-2.26.1	616	104
exiv2-0.25	1020	38

fuzzing. Further combining with the small stuck time periods, we could see that the fuzzer in a stuck state does not necessarily mean it actually needs concolic execution since most of the stuck states are really short (85 percent of the stuck states are under 100 seconds). These observations break Assumption 1 described above.

Second, the "demand launch" strategy cannot recognize the specific paths that block fuzzing, rendering very low effectiveness for concolic execution. On one hand, concolic execution takes 1654 seconds on average to process one input. On the other hand, a fuzzer often retains much more inputs than what concolic execution could handle. As a result, the input corresponding to the specific branch that blocks the fuzzing (i.e., the input that could lead execution to the target place) only has a very small chance to be picked up and processed by concolic execution. Therefore, Assumption 2 described above does not really hold in practice. This conclusion can be further confirmed the weak assistance from fuzzing where the concolic execution can help the fuzzing on merely 13 binaries despite that it is launched on 49 binaries. Moreover, only 51 inputs from the concolic execution are imported by the fuzzer after 1709 runs of concolic execution, indicating a very low quality of the inputs generated by concolic execution.

3 Probabilistic Path Prioritization Guided by Monte-Carlo

To address the aforementioned limitations of the current hybrid fuzzing systems, we propose a novel "discriminative dispatch" strategy to better combine fuzzing and concolic execution.

3.1 Key Challenge

As discussed above, the key challenge of our strategy is to quantify the difficulty of traversing a path for a fuzzer in a lightweight fashion. There are solutions for quantifying the difficulty of a path using expensive program analysis, such as value analysis [44] and probabilistic symbolic execution [5]. However, these techniques do not solve our problem: if we have already performed a heavyweight analysis to quantify the difficulty, we might as well just solve the path constraints and generate an input to traverse the path.

3.2 Monte Carlo Based Probabilistic Path Prioritization Model

In this study, we propose a novel Monte Carlo based Probabilistic Path Prioritization Model (MCP^3 for short) to deal with the challenge. In order to be lightweight, our model applies the Monte Carlo method to calculate the probability of a path to be explored by fuzzing. For the Monte Carlo method to work effectively, two requirements need to be full-filled: 1). the sampling to the search space has to be random; 2). a large number of random sampling is required to make the estimations statistically meaningful. Since a fuzzer randomly generates inputs for testing programs, our insight is to consider the executions on these inputs as random samples to the whole program state space, thus the first requirement is satisfied. Also, as fuzzing has a very high throughput to generate test inputs and perform executions, the second requirement can also be met. Therefore, by regarding fuzzing as a sampling process, we can statistically estimate the probability in a lightweight fashion with coverage statistics. A concern is the Monte Carlo method expects uniformly random samples, whereas grey-box fuzzers are generally not truly uniformly randomized in terms of seed mutation. Then, the probabilities calculated could also be biased. We present the discussion in Section 6.2.

According to the Monte Carlo method, we can simply estimate the probability of a path by statistically calculating the ratio of executions traversing this path to all the executions. However, this intuitive approach is not practical, because maintaining path coverage is a challenging and heavyweight task. With this concern, most of the current fuzzing techniques adopt a lightweight coverage metric such as block coverage and branch coverage. For this challenge, we treat an execution path as a Markov Chain of successive branches, inspired by a previous technique [4]. Then, the probability of a path can be calculated based on the probabilities of all the branches within the path.

Probability for Each Branch. The probability of a branch quantifies the difficulty for a fuzzer to pass a condition check and explore the branch. Equation 1 shows how MCP^3 calculates the probability of a branch.

$$P(br_i) = \begin{cases} \frac{cov(br_i)}{cov(br_i)+cov(br_j)}, & cov(br_i) \neq 0 \\ \frac{3}{cov(br_j)}, & cov(br_i) = 0 \end{cases} \quad (1)$$

In Equation (1), br_i and br_j are two branches that share the same predecessor block, and $cov(br_i)$ and $cov(br_j)$ refer to the coverage statistics of br_i and br_j, representing how many times br_i and br_j are covered by the samples from a fuzzer respectively.

When br_i has been explored by fuzzing ($cov(br_i)$ is non-zero), the probability for br_i can be calculated as the coverage statistics of br_i divided by the total coverage statistics of br_i and br_j.

When br_i has never been explored before ($cov(br_i)$ is zero), we deploy the *rule of three* in statistics [42] to calculate the probability of br_i. The *rule of three* states that if a certain event did not occur in a sample with n subjects, the interval from 0 to $3/n$ is a 95 percent confidence interval for the rate of occurrences in the population. When n is greater than 30, this is a good approximation of results from more sensitive tests. Following this rule, the probability of br_i becomes $3/cov(br_j)$ if $cov(br_j)$ is larger than 30. If $cov(br_j)$ is less than 30, the probability is not statistically meaningful. That is, we will not calculate the probabilities until the coverage statistics are larger than 30.

Probability for Each Path. To calculate the probability for a path, we apply the Markov Chain model [21] by viewing a path as continuous transitions among successive branches [4]. The probability for a fuzzer to explore a path is calculated as Equation (2).

$$P(path_j) = \prod \{P(br_i) | br_i \in path_j\}. \quad (2)$$

The $path_j$ in Equation (2) represents a path, br_i refers to a branch covered by the path and $P(br_i)$ refers the probability

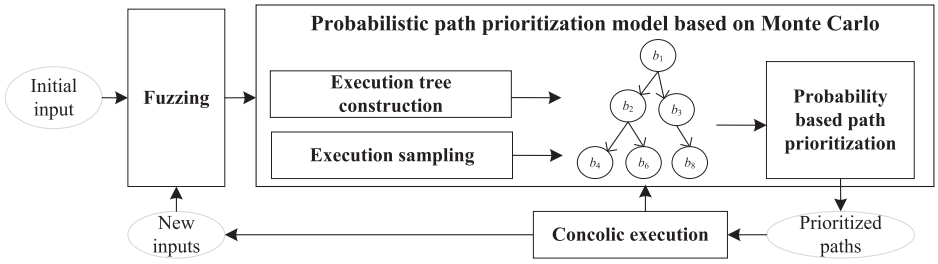


Fig. 4. Overview of DigFuzz.

of br_i. The probability of $path_j$ shown as $P(path_j)$ is calculated by multiplying the probabilities of all branches along the path together.

3.3 MCP^3-Based Execution Tree

In our "discriminative dispatch" strategy, the key idea is to infer and prioritize paths for concolic execution from the runtime information of executions performed by fuzzing. For this purpose, we construct and maintain a MCP^3 based execution tree, which is defined as follows:

Definition 1. An MCP^3 based execution tree is a directed tree $T = (V, E, \alpha)$, where:

- Each element v in the set of vertices V corresponds to a unique basic block in the program trace during an execution;
- Each element e in the set of edges $E \subseteq V \times V$ corresponds to the a *control flow dependency* between two vertices v and w, where $v, w \in V$. One vertex can have two outgoing edges if it contains a condition check;
- The labeling function $\alpha : E \rightarrow \Sigma$ associates edges with the labels of probabilities, where each label indicates the probability for a fuzzer to pass through the branch.

4 Design and Implementation

In this section, we present the system design and implementation details for DigFuzz.

4.1 System Overview

Fig. 4 shows an overview of DigFuzz. It consists of three major components: 1) a fuzzer; 2) the MCP^3 model; and 3) a concolic executor.

Our system leverages a popular off-the-shelf fuzzer, American Fuzzy Lop (AFL) [47] as the fuzzing component, and builds the concolic executor on top of *angr* [37], an open-source symbolic execution engine, the same as Driller [38].

The most important component in DigFuzz is the MCP^3 model. This component performs execution sampling, constructs the MCP^3 based execution tree, prioritizes paths based on the probability calculation, and eventually feeds the prioritized paths to the concolic executor.

DigFuzz starts the testing by fuzzing with initial seed inputs. As long as inputs are generated by the fuzzer, the MCP^3 model performs execution sampling to collect coverage statistics which indicate how many times each branch is covered during the sampling. Simultaneously, it also constructs the MCP^3 based execution tree through trace analysis and labels the tree with the probabilities for all branches that are calculated from the coverage statistics. Once the tree is constructed and paths are labeled with probabilities, the MCP^3 model prioritizes all the missed paths in the tree, and identifies the paths with the lowest probability for concolic execution.

As concolic execution simultaneously executes programs on both concrete and symbolic values for simplifying path constraints, once a missed path is prioritized, the MCP^3 model will also identify a corresponding input that can guide the concolic execution to reach the missed path. That is, by taking the input as a concrete value, the concolic executor can execute the program along with the prefix of the missed path, generate and collect symbolic path constraints. When reaching to the missed branch, it can generate the constraints for the missed path by conjoining the constraints for the path prefix with the condition to this missed branch. Finally, the concolic executor generates inputs for missed paths and feeds the generated inputs back to the fuzzer. Meanwhile, it also updates the execution tree with the paths that have been explored during concolic execution. By leveraging the new inputs from the concolic execution, the fuzzer will be able to move deeper, extent code coverage and update the execution tree.

To sum up, DigFuzz works iteratively. In each iteration, the MCP^3 model updates the execution tree through trace analysis on all the inputs retained by the fuzzer. Then, this model labels every branch with its probability that is calculated with coverage statistics on execution samples. Later, the MCP^3 model prioritizes all missed paths, and selects the path with the lowest probability for concolic execution. The concolic executor will generate inputs for the missed path, return the generated inputs to the fuzzer, and update the execution tree with paths that have been explored during concolic execution. After these steps, DigFuzz will enter into another iteration.

4.2 Execution Sampling

Random sampling is required for DigFuzz to calculate probabilities using the Monte Carlo method [34]. Based on the observation that a fuzzer by nature generates inputs randomly, we consider the fuzzing process as a random sampling process to the whole program state space. Due to the high throughput of fuzzing, the number of generated samples will quickly become large enough to be statistically meaningful, which is defined by *rule of three* [42] where the interval from 0 to 3/n is a 95 percent confidence interval when the number of samples is greater than 30.

Following this observation, we present Algorithm 1 to perform the sampling. This algorithm accepts three inputs and produces the coverage statistics in a $HashMap$. The three inputs are: 1) the target binary P; 2) the fuzzer $Fuzzer$; and 3) the initial seeds stored in Set_{inputs}. Given the three

```
     void main(argv) {                 int chk_in () {
       recv(in);            b6          res = is_valid(in)
       switch (argv[1]) {   b7          if (!res)
b1       case 'A':          b8             return;
b2         chk_in(in);      b9          if (strcmp(in, 'BK') == 0);
           break;           b10            //vulnerability
b3       case 'B':          b11         else …   }
b4         is_valid(in);              int is_valid(in) {
           break;           b12         if all_char(in)
b5       default: …         b13            return 1;
     }}                     b14         return 0; }
```

Fig. 5. Running example.

inputs, the algorithm iteratively performs the sampling during fuzzing. $Fuzzer$ takes P and Set_{inputs} to generate new inputs as $Set_{NewInputs}$ (Ln. 7). Then, for each input in $NewInputs$, we collect coverage statistical information for each branch within the path determined by P and $input$ (Ln. 9) and further update the existing coverage statistics stored in $HashMap_{CovStat}$ (Ln. 11 and 12). In the end, the algorithm merges $Set_{NewInputs}$ into Set_{inputs} (Ln. 15) and starts a new iteration.

Algorithm 1. Execution Sampling

```
1: P ← {Target binary}
2: Fuzzer ← {Fuzzer in DigFuzz }
3: Set_inputs ← {Initial seeds}
4: HashMap_CovStat ← ∅; Set_NewInputs ← ∅
5:
6: while True do
7:   Set_NewInputs ← Fuzzer{P, Set_inputs}
8:   for input ∈ Set_NewInputs do
9:     Coverage ← GetCoverage(P, input)
10:    for branch ∈ Coverage do
11:      Index ← Hash(branch)
12:      HashMap_CovStat{Index} + +
13:    end for
14:  end for
15:  Set_inputs ← Set_inputs ∪ Set_NewInputs
16: end while
    output the HashMap_CovStat as coverage statistics
```

4.3 Execution Tree Construction

As shown in Fig. 4, DigFuzz generates the MCP^3 based execution tree using the runtime information from fuzzer.

Algorithm 2 demonstrates the tree construction process. The algorithm takes three inputs, the control-flow graph for the target binary CFG, inputs retained by the fuzzer Set_{inputs} and the coverage statistics $HashMap_{CovStat}$, which is the output from Algorithm 1. The output is a MCP^3 based execution tree $ExecTree$. There are mainly two steps in the algorithm. The first step is to perform trace analysis for each $input$ in Set_{inputs} to extract the corresponding trace and then merge the trace into $ExecTree$ (Ln. 6 to 11). The second step is to calculate the probability for each branch in the execution tree (Ln. 12 to 16). To achieve this, for each branch br_i in $ExecTree$, we extract its neighbor branch br_j (br_i and br_j share the same predecessor block that contains a condition check) by examining the CFG (Ln. 13). Then, the algorithm leverages Equation 1 to calculate the probability for br_i (Ln. 14). After that, the algorithm labels $ExecTree$ with the calculated probabilities (Ln. 15) and outputs the newly labeled $ExecTree$.

Algorithm 2. Execution Tree Construction

```
1: CFG ← {Control flow graph for the target binary.}
2: Set_inputs ← {Inputs retained by the fuzzer}
3: HashMap_CovStat ← {Output from Algorithm 1}
4: ExecTree ← ∅
5:  6: for input ∈ Set_inputs do
7:   trace ← TraceAnalysis(input)
8:   if trace ∉ ExecTree then
9:     ExecTree ← ExecTree ∪ trace
10:  end if
11: end for
12: for br_i ∈ ExecTree do
13:   br_j ← GetNeighbor(br_i, CFG)
14:   prob ← CalBranchProb(br_i, br_j, HashMap_CovStat)
15:   LabelProb(ExecTree, br_i, prob)
16: end for
    output ExecTree as the MCP^3 based execution tree
```

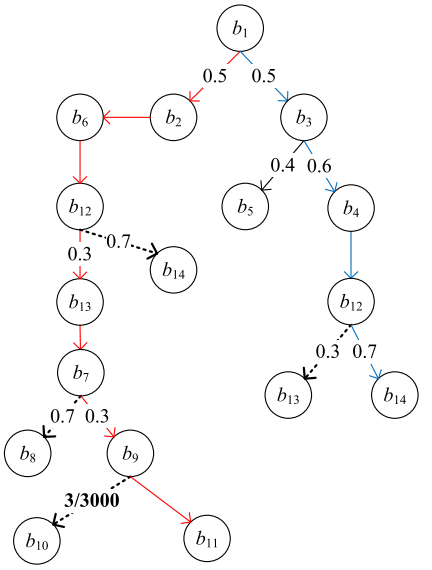
To avoid the potential problem of path explosion in the execution tree, we only perform trace analysis for the seed inputs retained by fuzzing. The fuzzer typically regards those mutated inputs with new code coverage as seeds for further mutation. Tracing on these retained seeds is a promising approach to model the explored program state. To be more specific, we only perform trace analysis for *interesting inputs* retained in AFL [47]. A mutated input is saved as an *interesting input* according to the metrics of both sizes and execution time.

For each branch along an execution trace, whenever the opposite branch has not been covered by fuzzing, then a missed path is identified, which refers to a prefix of the trace conjoined with the uncovered branch. In other words, the execution tree does not include an uncovered branch of which the opposite one has not been covered yet.

To ease representation, we present a running example, which is simplified from a program in the CQE dataset [15], and the code piece is shown in Fig. 5. The vulnerability is guarded by a specific string, which is hard for fuzzing to detect. Fig. 6 illustrates the MCP^3 based execution tree for the running example in Fig. 5. Each node represents a basic block. Each edge refers to a branch labeled with the probability. We can observe that there are two traces ($t_1 = \langle b_1, b_2, b_6, b_{12}, b_{13}, b_7, b_9, b_{11}\rangle$ and $t_2 = \langle b_1, b_3, b_4, b_{12}, b_{14}\rangle$) in the tree marked as red and blue. Note that the probabilities are calculated by multiple execution samples. In this example, we only present two traces for easing presentation.

4.4 Probabilistic Path Prioritization

We then prioritize paths based on probabilities. As shown in Equation (2), a path is treated as a Markov chain and its probability is calculated based on the probabilities of all the branches within the path. A path can be represented as a sequence of covered branches, and each branch is labeled with its probability that indicates how likely a random input can satisfy the condition. Consequently, we leverage the Markov Chain model to regard the probability for a path as the sequence of probabilities of the transitions.



P_1=<b_1, b_2, b_6, b_{12}, b_{13}, b_7, b_9, b_{10}>, $P(P_1)$=0.000045
P_2=<b_1, b_2, b_6, b_{12}, b_{13}, b_7, b_8>, $P(P_2)$=0.105
P_3=<b_1, b_2, b_6, b_{12}, b_{14}>), $P(P_3)$=0.35
P_4=<b_1, b_3, b_4, b_{12}, b_{13}>, $P(P_4)$=0.09

Fig. 6. The execution tree with probabilities.

Algorithm 3. Path Prioritization in DigFuzz

1: $ExecTree \leftarrow$ {Output from Algorithm 2}
2: $Set_{Prob} \leftarrow \emptyset$
3: **for** $trace \in ExecTree$ **do**
4: **for** $br_i \in trace$ **do**
5: $br_j \leftarrow$ GetNeighbor(br_i, CFG)
6: $missed \leftarrow$ GetMissedPath($trace, br_i, br_j$)
7: **if** $missed \notin ExecTree$ **then**
8: $prob \leftarrow$ CalPathProb($missed$)
9: $Set_{Prob} \leftarrow \{trace, missed, prob\}$
10: **end if**
11: **end for**
12: **end for**
output Set_{Prob} as missed paths with probabilities corresponding to each trace

The detailed algorithm is presented in Algorithm 3. It takes the MCP^3 based execution tree $ExecTree$ from Algorithm 2 as the input and outputs Set_{Prob}, a set of missed paths, and their probabilities. Our approach will further prioritize these missed paths based on Set_{Prob} and feed the one with the lowest probability to concolic execution. The algorithm starts with the execution tree traversal. For each branch br_i on every $trace$ within $ExecTree$, it first extracts the neighbor br_j (Ln. 5) and then collects the missed paths $missed$ along the given trace (Ln. 6). Then, the algorithm calculates the probability for $missed$ by calling $CalPathProb()$ which implements Equation (2) and stores the information in Set_{Prob}. Eventually, the algorithm produces Set_{Prob}, a set of missed paths with probabilities for every trace.

After we get Set_{Prob}, we will prioritize missed paths by a decrease order on their probabilities, and identify the path with the lowest probability for concolic execution. As the concolic executor takes a concrete input as the concrete value to perform trace-based symbolic execution, we will identify an input on which the execution is able to guide the concolic executor to the prioritized missed path.

Take the program in Fig. 5 as an example. In Fig. 6, the missed branches are shown as dotted lines. After the execution tree is constructed and properly labeled, Algorithm 3 is used to obtain missed paths and calculate probabilities for these paths. We can observe that there are 4 missed paths in total denoted as P_1, P_2, P_3, and P_4, respectively. By calling *CalPathProb()* function, the probabilities of these missed paths are calculated as shown in the figure, and the lowest one is of P_1. To guide the concolic executor to explore P_1, our approach will pick the input that leads to the trace $\langle b_1, b_2, b_6, b_{12}, b_{13}, b_7, b_9, b_{11}\rangle$ and assign this input as the concrete value of concolic execution, because this trace share the same path prefix, $\langle b_1, b_2, b_6, b_{12}, b_{13}, b_7, b_9\rangle$, with the missed path P_1.

5 Evaluation

In this section, we conduct a comprehensive evaluation on the performance of DigFuzz by comparing it with state-of-the-art hybrid fuzzing systems, with respect to code coverage, the number of discovered vulnerabilities, and the contribution of concolic execution.

We build DigFuzz on top of a popular fuzzer, American Fuzzy Lop (AFL) [47], and two open-source concolic executors, *angr* [37] and *QSYM* [46]. To evaluate our system against DARPA Cyber Grand Challenge (CGC) binaries [15], we integrate *angr* [37], which is designed to support specific syscalls in the CGC customized platform. We further integrate *QSYM* [46] into our system, because *angr* [37] has insufficient syscalls support for real-world programs. Therefore, we employ *angr* [37] and *QSYM* [46] to analyze CGC binaries and real-world programs respectively.

5.1 Datasets

We leverage three datasets: the CGC Qualifying Event (CQE) dataset [15] (same as in Driller [38]), the `LAVA-M` dataset and 12 real-world Linux programs.

The CQE dataset contains 131 applications that are deliberately designed by security experts to test automated vulnerability detection systems. Every binary is injected with one or more memory corruption vulnerabilities. In addition, many CQE binaries have complex protocols and large input spaces, making these vulnerabilities harder to trigger. In our evaluation, we exclude 5 applications involving communication between multiple binaries as in Driller, and 8 applications on which AFL cannot work. Totally, we use 118 CQE binaries for evaluation.

For the LAVA dataset [17], we adopt `LAVA-M` as previous techniques [11], [26], which consists of 4 real-world programs, `uniq`, `base64`, `md5sum`, and `who`. Each program in `LAVA-M` is injected with multiple bugs, and each injected bug has a unique ID.

We choose Linux programs based on the following features: popularity, frequency of being tested, and diversity of categories. These 12 Linux programs (as shown in Table 3) include well-known tools (e.g., *nm*, *objdump*), image processing libraries (e.g., *libjpeg*, *libtiff*), terminal processing libraries (e.g., *ncurses*,), and document processing libraries (e.g., *xpdf*), etc. As shown in Table 3, the sizes of these 12 Linux

TABLE 3
Specific Information About the 12 Linux Programs

Binaries & Versions	Sizes(KB)	Input format	Test command
libjpeg-9b+cjpeg	692	jpg	cjpeg @@
ncurses-6.1+infotocap	253	text	infotocap @@
jhead-3.00	85	jpg	jhead @@
nm-2.26.1	5791	elf	nm -AD @@
xpdf-4.00+pdfimages	1677	pdf	pdfimages @@ /dev/null
libpng-1.6.36+pngfix	978	png	pngfix @@
readelf-2.26.1	474	elf	readelf -a @@
size-2.26.1	5745	elf	size -At @@
libtiff-4.0.10+tiffdump	23	tiff	tiffdump @@
libxpat-R.2.2.5+xmlwf	258	xml	xmlwf @@
objdump-2.26.1	349	elf	objdump -fdDh @@
exiv2-0.25	3147	jpg	exiv2 @@ /dev/null

@@ is a command in AFL, which indicates that the input is a file.

programs range from 23KB to 5791KB. The initial seeds for all these 12 Linux programs are defined as the seed examples provided by AFL. We regard the unique crashes reported by AFL as the detected crashes for these 12 Linux programs.

5.2 Baseline Techniques

As the main contribution of DigFuzz is to propose a more effective strategy to combine fuzzing with concolic execution, the advance of fuzzing itself is out of our scope. Therefore, we do not compare DigFuzz with non-hybrid fuzzing systems such as CollAFL [18], Angora [11], AFLfast [4], and VUzzer [33].

To quantify the contribution of concolic execution, we leverage pure fuzzing as a baseline. We deploy the original AFL to simulate fuzzing assisted by a dummy concolic executor that makes zero contribution. This configuration is denoted as AFL.

To compare DigFuzz with other hybrid fuzzing systems, we choose the state-of-the-art hybrid fuzzing system, Driller, QSYM [46], and MDPC.

We use Driller$*$ [1] to represent the configuration of Driller. Moreover, to evaluate the impact of the path prioritization component alone, we modify Driller to enable the concolic executor to start from the beginning by randomly selecting inputs from fuzzing. We denote this configuration as Random. The only difference between Random and DigFuzz is the path prioritization. This configuration eliminates the first limitation described in Section 2.2.

QSYM [46] mainly improves the performance of concolic execution by optimizing emulation speed and reducing emulation usage. Our research focuses on the path prioritization for hybrid fuzzing, thus we only evaluate the strategy of path prioritization in *QSYM* (denoted as QSYM) for a fair comparison. That is, baseline techniques, such as Driller$*$ and QSYM, only differ on the strategies of path prioritization. QSYM prioritizes paths for concolic execution based on the sizes and times of generated inputs from fuzzing. To be more specific, it prefers to select smaller inputs for concolic execution. If the sizes of two inputs are the same, QSYM will further select the newer one according to their generated time by the fuzzer.

1. The configuration of Driller$*$ in our work is different from Driller paper as further discussed in Section 5.3.

In the original MDPC model [41], fuzzing and concolic execution alternate in a sequential manner, whereas all the other hybrid systems work in parallel to better utilize computing resources. To make a fair comparison, we configure the MDPC model to work in parallel. More specifically, if MDPC chooses fuzzing to explore a path, then the fuzzer generates a new test case for concrete testing. Otherwise, MDPC will assign this path that requires to be explored by concolic execution to a job queue, and continues to calculate probabilities for other paths. The concolic executor will take a path subsequently from the job queue. In this way, we can compare MDPC with other hybrid systems with the same computing resources.

Besides, as estimating the cost for solving a path constraint is a challenging problem, we simply assign every path constraints with the solving cost of 50, which is the highest solving cost as defined [41]. Please note that with this configuration, the time cost by MDPC for optimal decision is lower, because it does not spend effort in collecting and estimating path constraints.

We did not take SAVIOR [13] as a baseline technique for several reasons. First, the two techniques follow totally different design principles. DigFuzz is a code coverage-driven testing, whereas SAVIOR adopts a bug-driven approach. Our work aims to demonstrate the effectiveness of our path prioritization strategy in terms of code coverage. Therefore, code coverage, which is the main metric to evaluate the effectiveness of DigFuzz, is completely pointless for SAVIOR. Second, SAVIOR [13] requires source code in order to perform static analysis to find potential bugs. In contrast, DigFuzz works with pure binaries. Besides, SAVIOR designs a customized concolic execution engine, based on KLEE [6], for addition security checking besides input generation. As a result, the two techniques embrace different deployment and threat models. Third, for a relatively fair comparison, we tried our best to evaluate only the bug-driven path prioritization in SAVIOR, by disabling the security check in the customized KLEE. However, the customized KLEE is currently not open-source. We contacted with the authors of SAVIOR and confirmed that the current version of SAVIOR can only work in their prebuilt docker. Consequently, there is no way we can modify the tool and disable the security check for a fair comparison with DigFuzz.

5.3 Experiment Setup

The original Driller [38] adopts a shared pool design, where the concolic execution pool is shared among all the fuzzer instances. With this design, when a fuzzer gets stuck, Driller adds all the inputs retained by the fuzzer into a global queue of the concolic execution pool and performs concolic execution by going through these inputs sequentially. This design is not suitable for us as the fuzzer instances are not fairly aided by the concolic execution.

To better evaluate our new combination strategy in DigFuzz, we assign computer resources evenly to ensure that the analysis on each binary is fairly treated. As there exist two modes in the mutation algorithm of AFL (deterministic and non-deterministic modes), we allocate 2

fuzzing instances (each running in one mode) for every testing binary. In detail, we allocate 2 fuzzing instances for testing binaries with AFL, 2 fuzzing instances and 1 concolic execution instance with Driller∗, Random, QSYM, MDPC and DigFuzz. Each run of concolic execution is limited to 4GB of memory and run-time up to one hour, which is the same as in Driller.

We run the experiments on a server with three computer nodes, and each node has 18 CPU cores and 32GB RAM. Considering that random effects play an important role in our experiments, we choose to run each experiment for three times, and report the mean values for a more comprehensive understanding of the performance. In order to give enough time for fuzzing as well as limit the total time of three runs for each binary, we choose to assign 12 hours to each binary from the CQE dataset, and stop the analysis as long as a crash is observed. For the LAVA dataset, we analyze each binary for 5 hours as in the LAVA paper. For the Linux programs, we analyze each binary for 12 hours.

5.4 Evaluation on the CQE Dataset

In this section, we demonstrate the effectiveness of our approach on the CQE dataset from three aspects: code coverage, the number of discovered vulnerabilities, and the contribution of concolic execution to the hybrid fuzzing.

5.4.1 Code Coverage

Code coverage is a critical metric for evaluating the performance of a fuzzing system. We use the bitmap maintained by AFL to measure the code coverage. In AFL, each branch transition is mapped into an entry of the bitmap via hashing. If a branch transition is explored, the corresponding entry in the bitmap will be filled and the size of the bitmap will increase.

As the program structures vary from one binary to another, the bitmap sizes of different binaries are not directly comparable. Therefore, we introduce a metric called *normalized bitmap size* to summarize how the code coverage increases for all tested binaries. For each binary, we treat the code coverage of the initial inputs as the base. Then, at a certain point during the analysis, the *normalized bitmap size* is calculated as the size of the current bitmap divided by the base. This metric represents the increasing rate of the bitmap.

Fig. 7 presents how the average of *normalized bitmap size* for all binaries grows. The figure shows that DigFuzz constantly outperforms the other fuzzing systems. By the end of the 12-hours testing, the *normalized bitmap sizes* in DigFuzz, Random, Driller∗, QSYM, and AFL are 3.46 times, 3.25 times, 3.02 times, 3.01 times, and 2.91 times larger than the base, respectively. Taking the *normalized bitmap sizes* in AFL that is aided by dummy concolic execution as a baseline, the concolic execution in DigFuzz, Random, Driller∗, and QSYM contribute to discovering 18.9 percent, 11.7, 3.8, and 3.4 percent more code coverage, respectively. To address the concern that the average coverage might be tweaked by a few programs with high variances, we further calculate the number of programs on which DigFuzz outperforms other techniques in terms of code coverage.

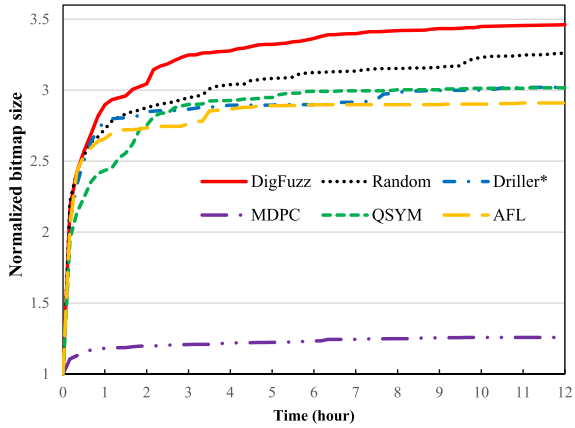


Fig. 7. Normalized bitmap size on CQE dataset.

Specifically, DigFuzz outperforms other techniques on 97 out of 118 binaries by the end of the 12-hours testing.

We can draw several conclusions from the numbers. First, Driller∗ can considerably outperform AFL. This indicates that concolic execution could indeed help the fuzzer. This conclusion is aligned with the Driller paper [38]. Second, the optimization in Random does help increase the effectiveness of the concolic execution compared to Driller∗. This observation shows that the second limitation of the "demand launch" strategy described in Section 2 can considerably affect the concolic execution. Third, by comparing DigFuzz with QSYM and Random, we can observe that the path prioritization implemented in DigFuzz greatly strengthens the hybrid fuzzing system in exploring new branches. Further investigation shows that the contribution of concolic execution to bitmap size in DigFuzz is much larger than those in Driller∗ (18.9 versus 3.8 percent), QSYM (18.9 versus 3.4 percent), and Random (18.9 versus 11.7 percent). This fact demonstrates the effectiveness of our strategy in terms of code exploration.

We can also see that MDPC is even worse than AFL. By carefully examining the working progress of MDPC, we find that the main reason is the reduced throughput of fuzzing. In contrast to the average throughput in AFL that is 417 executions per second, the throughput reduces to 2.6 executions per second. It indicates that decision making for each path as MDPC is too expensive, completely taking away the power of fuzzing.

As an optimization, one would move the optimal decision module out and make it work in parallel with fuzzing. In this manner, the concurrent MDPC would be able to take advantage of the high throughput of fuzzing. However, using the defined solving cost [41], the concurrent MDPC assigns all the missed paths to concolic execution only in several seconds after the fuzzing starts. Then, the concurrent MDPC will degrade to Random. The reason is that the cost of concolic execution (50 as defined in the original paper) might be too small. Actually, how to normalize the cost of fuzzing and concolic for a unified comparison is difficult, because these two costs are estimated by using different metrics, which are concerned with the run-time throughput of fuzzing, the performance of the constraint solver, and the symbolic execution engine. It is difficult (if not impossible) to define a unified metric to evaluate the costs of different techniques.

TABLE 4
Number of Discovered Vulnerabilities

	= 3	≥ 2	≥ 1
DigFuzz	73	77	81
Random	68	73	77
Driller∗	67	71	75
QSYM	65	70	73
MDPC	29	29	31
AFL	68	70	73

Unlike MDPC that estimates the costs for exploring each path by fuzzing and concolic execution respectively, DigFuzz prioritizes paths by quantifying the difficulties for fuzzing to explore a path based on coverage statistics. Granted, the sampling may be biased as generated test cases by fuzzing are not uniform distributed rendering even lower possibility to explore the difficult paths than theory, such bias in fact works for our favor. Our goal is to find the most difficult branches for fuzzing by quantifying the probabilities. With the bias, it lowers the probability calculated and increases the chance for DigFuzz to pick the least visited branches by fuzzing.

5.4.2 Number of Discovered Vulnerabilities

Table 4 presents the numbers of vulnerabilities discovered by all four configurations. Column 2 displays the numbers of vulnerabilities discovered in all three runs. Similarly, columns 3 and 4 show the numbers of vulnerabilities that are discovered at least twice and once out of three runs, respectively.

We can observe that in all three metrics, DigFuzz discovers considerably more vulnerabilities than the other configurations. In contrast, Driller∗ only has a marginal improvement over AFL. Random discovers more vulnerabilities than Driller∗ yet still falls behind DigFuzz due to the lack of path prioritization. Another interesting observation is that QSYM discovers less vulnerabilities than other configurations, and even less than AFL. It indicates that the fuzzing in QSYM is not well aided by concolic execution. This result further suggests that the size and temporal order are not good indicators for path prioritization.

This table could further exhibit the effectiveness of DigFuzz by comparing it with the numbers reported in the Driller paper. In the paper, Driller assigns 4 fuzzing instances for each binary, and triggers crashes in 77 applications in 24 hours [38]. Among these 77 binaries, 68 of them are crashed purely by AFL, and only 9 binaries are crashed with the help of concolic execution. This result is on par with the numbers in column 3 for DigFuzz. This means DigFuzz is able to perform similarly with only half of the running time (12 hours versus 24 hours) and much fewer hardware resources (2 fuzzing instances per binary versus 4 fuzzing instances per binary).

5.4.3 Contribution of Concolic Execution

Here, we dive deeper into the contribution of concolic execution by presenting some detailed numbers for imported inputs and crashes derived from the concolic execution.

TABLE 5
Performance of Concolic Execution on CQE

	Ink.	CE	Aid.	Imp.	Der.	Vul.
DigFuzz	64	1251	37	719	9,228	12
	64	668	39	551	7,549	11
	63	1110	41	646	6,941	9
Random	68	1519	32	417	5,463	8
	65	1235	23	538	5,297	6
	64	1759	21	504	6,806	4
Driller*	48	1551	13	153	1,679	5
	49	1709	12	51	2,375	4
	51	877	13	95	1,905	4
QSYM	69	934	34	576	9,641	4
	68	1267	35	513	8,149	4
	68	1015	34	562	9,130	4

An important metric for evaluating the contribution of concolic execution is how many inputs generated by concolic execution are then imported by the fuzzer. In the implementation, AFL marks down the inheritance relations between inputs through the unique ID and source ID of inputs. With this inheritance information, we can analyze how many inputs are imported by AFL. Furthermore, some of these imported inputs lead to crashes.

Table 5 presents these details for DigFuzz, Random, and Driller∗, each for three runs. The second column (Ink.) lists the number of binaries for which the concolic execution is invoked during testing. For this number, we exclude binaries on which the concolic execution is invalid. Such invalid concolic execution is either caused by path divergence that the symbolic execution path disagrees with the realistic execution trace, or caused by the resource limitation (we kill the concolic execution running for more than 1 hour or exhausting more than 4GB memory).

The third column (CE) shows the total number of concolic executions launched on all the binaries. We can see that Random invokes slightly more concolic execution jobs than DigFuzz, indicating that a concolic execution job in DigFuzz takes a bit longer to finish. As the fuzzing keeps running, the specific branches that block fuzzing become deeper. This result implies that DigFuzz is able to select high quality inputs and dig deeper into the paths.

The forth column (Aid.) refers to the number of binaries on which the fuzzer imports at least one generated input from the concolic execution. We can observe that the number for Random is larger than that in Driller∗. This indicates that the concolic execution can better help the fuzzer if it is invoked from the beginning. This result also confirms that a non-stuck state of the fuzzer does not necessarily mean the concolic execution is not needed. Further, since the number for DigFuzz is larger than Random, it shows that the concolic execution can indeed contribute more with path prioritization.

The fifth column (Imp.) refers to the number of inputs that are imported by the fuzzer from concolic execution while the sixth column (Der.) shows the number of inputs derived from those imported inputs by the fuzzer. We can see significant improvements on imported inputs and derived inputs for DigFuzz than Random and Driller∗. These improvements show that the inputs generated by DigFuzz are of much better quality in general. We can see

TABLE 6
Number of Discovered Vulnerabilities

Binaries	With dictionaries						Without dictionary					
	DigFuzz	Random	Driller∗	QSYM	MDPC	AFL	DigFuzz	Random	Driller∗	QSYM	MDPC	AFL
base64	48	48	48	47	48	32	3	3	3	3	3	2
md5sum	59	59	59	59	13	58	0	0	0	0	0	0
uniq	28	28	25	28	2	29	0	0	0	0	0	0
who	167	153	142	138	39	125	111	92	26	103	34	0

significantly more imported inputs and derived inputs contributed by the concolic execution component in DigFuzz than in Random and in Driller∗, even though fewer concolic execution jobs are invoked in DigFuzz.

The last column (Vul.) shows the number of binaries for which the crashes are derived from concolic execution. For each crashed binary, we identify the input that triggers the crash, and then examine whether the input is derived from an imported input generated by the concolic execution. The number shows that concolic execution in DigFuzz contributes to discovering more crashes (12 versus 5) than that in Driller∗.

To sum up, from the numbers reported, we clearly see that by mitigating the two limitations of the "demand launch" strategy, our new strategy outperforms the state-of-the-art hybrid system, Driller, in every important aspect.

5.5 Evaluation on the LAVA Dataset

In this section, we demonstrate the effectiveness of our approach on the LAVA-M dataset, which is widely adopted in recent studies [11], [26], [31].

5.5.1 Discovered Vulnerabilities

A recent report [16] shows that the fuzzing on binaries in LAVA-M can be assisted by extracting constants from these binaries and constructing dictionaries for AFL. We analyze every binary for 5 hours with and without dictionaries respectively.

The discovered vulnerabilities are shown in Table 6. It shows that with dictionaries, the five techniques, DigFuzz, Random, Driller∗, QSYM, and AFL can detect nearly all injected bugs in $base64$, $md5sum$ and $uniq$. With the impact of reduced of throughput, MDPC discovers less vulnerabilities than other techniques. By contrast, without dictionaries, these techniques can detect significantly fewer bugs (and in many cases, no bug). As demonstrated in LAVA [17], the reasons why concolic execution cannot make any contribution for $md5sum$ and $uniq$ are hash functions and unconstrained control dependency. These results indicate that it is the dictionaries contributing to detect most bugs in $base64$, $md5sum$, and $uniq$.

An exception is who, for which, DigFuzz outperforms Random, Driller∗, and AFL with a large margin. Looking closer, Driller∗ can only detect 26 bugs in who, while DigFuzz could detect 111 bugs. To better understand the result, we carefully examine the whole testing process, and find that the concolic executor in Driller∗ is less invoked than that in DigFuzz and Random. This shows even the fuzzer in Driller∗ could not make much progress in finding bugs, it rarely gets stuck when testing who. This result confirms our claim that the stuck state is not a good indicator for launching concolic execution and a non-stuck state does not necessarily mean concolic execution is not needed.

An exception is who, for which, DigFuzz outperforms Random, Driller∗, QSYM, MDPC, and AFL with a large margin. Looking closer, Driller∗ can only detect 26 bugs in who, MDPC can detect 34 bugs, while DigFuzz could detect 111 bugs. To better understand the result, we carefully examine the whole testing process, and find that the concolic executor in Driller∗ is less invoked than that in DigFuzz and Random. This shows even the fuzzer in Driller∗ could not make much progress in finding bugs, it rarely gets stuck when testing who. This result confirms our claim that the stuck state is not a good indicator for launching concolic execution and a non-stuck state does not necessarily mean concolic execution is not needed. Likewise, with the impact of reduced throughput, the fuzzer in MDPC generates fewer seeds than DigFuzz and Random. Then the number of execution paths on these seeds will be smaller as well. That is, the task of path exploration for the concolic executor in MDPC is lighter than concolic executors in DigFuzz and Random. As a consequence, MDPC explores smaller program states and discovers fewer bugs than DigFuzz and Random.

5.5.2 Code Coverage

As the trigger mechanism used by LAVA-M is quite simple (a comparison against a 4-byte magic number), extracting constants from binaries and constructing dictionaries for AFL will be very helpful [16], especially for $base64$, $md5sum$, and $uniq$. Consequently, the code coverage generated by these fuzzing systems will be about the same if dictionaries are used. As a result, we present the code coverage without dictionaries.

From Fig. 8, we can observe that DigFuzz can cover more code than the other three configurations. Both of the DigFuzz and Random outperform Driller∗, and all the hybrid systems is stably better than AFL.

The code coverage in Fig. 8 shows that our system is more effective than Random with only a very small margin. This is due to the fact that all the four programs are very small, and the injected bugs are close to each other. For example, in who, all the bugs are injected into only two functions. With these two factors, the execution trees generated from the programs in LAVA-M are small and contain only a few execution paths. Thus, path prioritization (the core part in DigFuzz) cannot contribute much since there exists no path explosion problem.

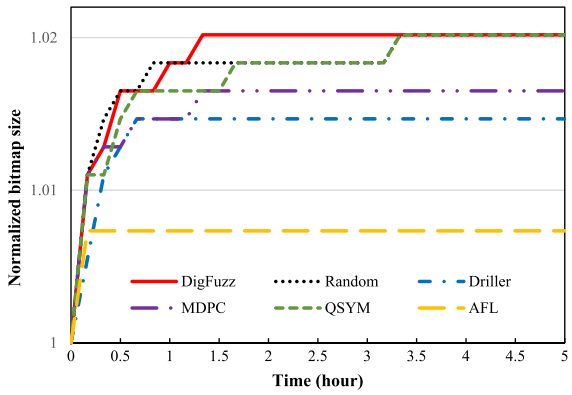


Fig. 8. Normalized incremental bitmap size on LAVA dataset.

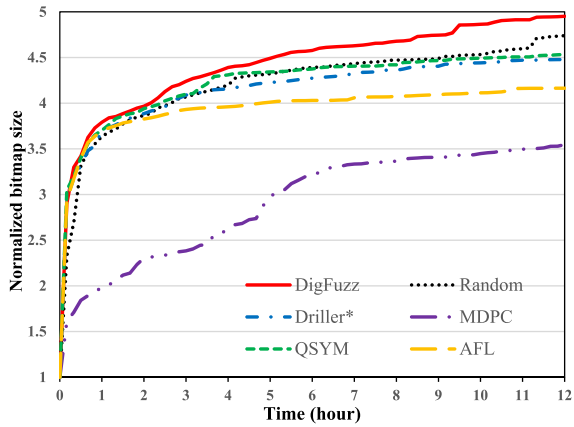


Fig. 9. Normalized bitmap size on the 12 Linux programs.

5.6 Evaluation on Real-World Programs

In this section, we demonstrate the effectiveness of our approach on real-world programs. In general, real-world programs are more complicated than CQE and LAVA-M binaries. According to the design principles, our "discriminative dispatch" strategy should enable more contributions from concolic executors.

5.6.1 Code Coverage

In this section, we use the *normalized bitmap size* as defined in Section 5.4.1 to summarize how the code coverage increases for all the 12 Linux binaries. Fig. 9 presents how the average of normalized bitmap size for all the 12 Linux binaries grows. By the end of 12 hours, the normalized bitmap sizes in DigFuzz, Random, Driller∗, QSYM and AFL are 4.95 times, 4.73 times, 4.47 times, 4.53 times, and 4.16 times larger than the base, respectively. Taking the normalized bitmap sizes in AFL that is aided by dummy concolic execution as a baseline, the concolic execution in DigFuzz, Random, Driller∗, and QSYM contribute to discovering 19.0, 13.7, 7.4, and 8.9 percent more code coverage, respectively.

These results confirms the following conclusions drawn from the results on CQE binaries: 1) concolic execution can indeed help the fuzzer; 2) the path prioritization implemented in DigFuzz strengthens the hybrid fuzzing system in exploring new branches. The contribution of concolic execution to bitmap size in DigFuzz is much larger than those in Driller∗ (19.0 versus 7.4 percent), QSYM (19.0 versus 8.9 percent), and Random (19.0 versus 13.7 percent). These results demonstrate the effectiveness of our strategy in terms of code exploration.

As an in-depth analysis, we present the increasing bitmap size for every program in Fig. 10. We can observe that DigFuzz explores more or at least the same code than Random, Driller∗, QSYM, MDPC, and AFL for all the 12 Linux programs.

5.6.2 Crashes

Table 7 present the average number of discovered crashes by different techniques.

We can clearly observe that DigFuzz discovers more crashes than other techniques, achieving a total of 178 unique crashes. In contrast, Driller∗ only has a marginal improvement over AFL (129 unique crashes versus 109 unique crashes). Random discovers more crashes than Driller∗ yet still falls behind DigFuzz due to the lack of path prioritization. QSYM discovers more crashes than Random. However, recall that QSYM discovers even less vulnerabilities than other techniques on the CQE binaries. We can infer that this strategy may vary from target programs.

By count the number of crashed binaries, DigFuzz discovers crashes in 7 out of the 12 binaries. In contrast, Random, Driller∗, QSYM, MDPC, and AFL discover crashes in 5, 3, 5, 1, and 3 binaries, respectively. We can further observe that all techniques (except MDPC) can discovery crashes in *libjpeg-9b+cjpeg*, *ncurses-6.1+infotocap*, and *objdump-2.26.1*. In addition, only DigFuzz discovers crashes in *readelf-2.26.1*. These comparisons can also demonstrate the effectiveness of DigFuzz.

5.6.3 Contribution of Concolic Execution

Besides the number of crashes, we present the contribution of concolic execution in Table 8. The first 4-columns (2-5) in Table 8 shows the number of concolic executions. The second 4-columns (6-9) shows the number of inputs that are imported by fuzzing from concolic execution. The last 4-columns (10-13) shows the number of inputs derived from those imported inputs by fuzzing.

From Table 8 we can observe that the concolic execution in Driller∗ is never invoked in *libtiff-4.0.10+tiffdump*. From the numbers of imported inputs and derived inputs for DigFuzz, Random, and QSYM, we can see that the concolic execution, in fact, can be very effective in helping fuzzing explore program space, since fuzzing actually imports hundreds of new inputs from concolic execution. This result further demonstrates that a non-stuck state in fuzzing does not necessarily mean concolic execution is not needed, which clearly breaks the first assumption in the 'demand launch' strategy. By comparing the total number of imported inputs in DigFuzz with those in other techniques, we can observe that concolic execution contributes more inputs.

In particular, for *ncurses-6.1+infotoca* and *libpng-1.6.36 +pngfix*, both the numbers of imported and derived inputs in QSYM are much larger than those in other techniques. These results indicate the concolic execution in QSYM makes more contribution in terms of imported and derived inputs.

To reason this result, we further perform manual analysis on *ncurses-6.1+infotoca* and *libpng-1.6.36+pngfix*, and find that a possible reason is that concolic executor fails to solve out path

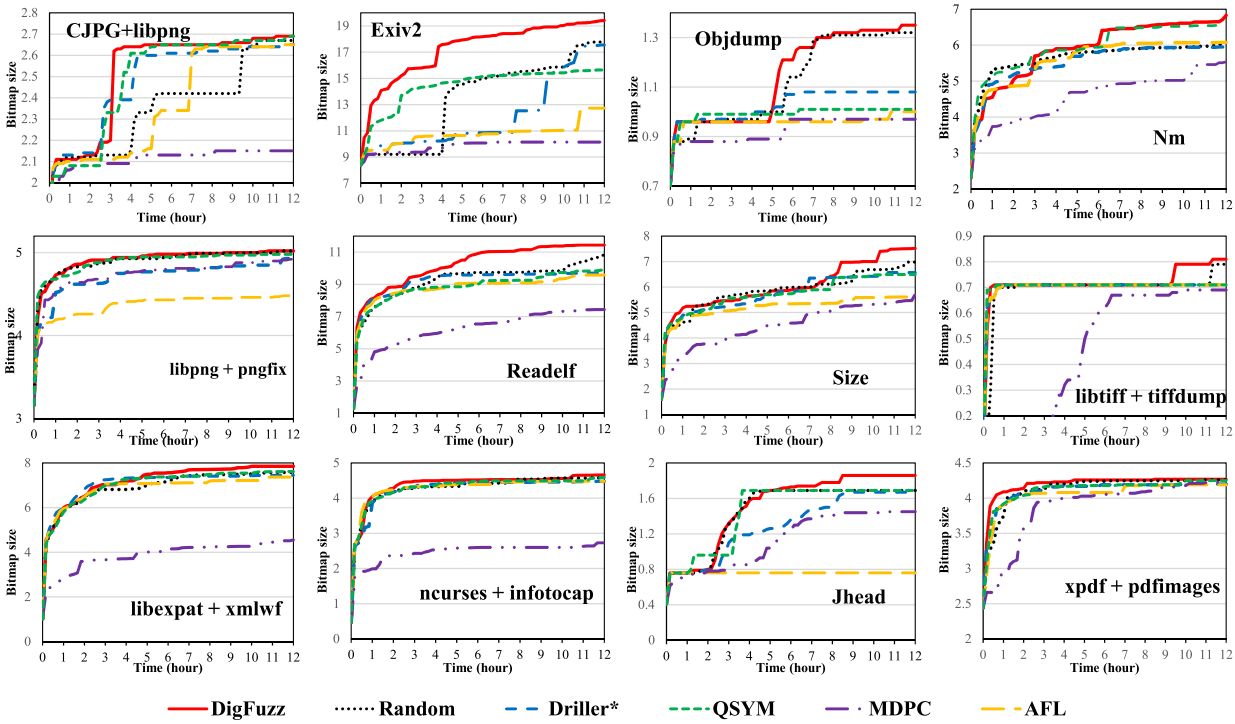


Fig. 10. Bitmap size on real-world programs.

TABLE 7
Discovered Crashes on Real-World Binaries

	DigFuzz	Random	Driller*	QSYM	MDPC	AFL
libjpeg-9b+cjpeg	31	21	24	29	0	15
ncurses-6.1+infotocap	125	110	96	119	0	86
jhead-3.00	1	0	0	1	0	0
nm-2.26.1	0	0	0	0	0	0
xpdf-4.00+pdfimages	1	1	0	0	0	0
libpng-1.6.36+pngfix	0	0	0	0	0	0
readelf-2.26.1	1	0	0	0	0	0
size-2.26.1	2	1	0	1	0	0
libtiff-4.0.10+tiffdump	0	0	0	0	0	0
libxpat-R.2.2.5+xmlwf	0	0	0	0	0	0
objdump-2.26.1	17	8	9	10	3	8
exiv2-0.25	0	0	0	0	0	0
Total	178	141	129	160	3	109

constraints. Take the hybrid fuzzing results on *libpng-1.6.36 +pngfix* for illustration. The concolic executor in DigFuzz performs 394 times of concolic execution, generates 797 inputs, and 204 of the 797 inputs are imported by the fuzzer. In contrast, the concolic executor in QSYM performs 386 times of concolic execution, which is less than that in DigFuzz. However, the concolic executor in QSYM generates 827 inputs and 227 of them are imported. These results indicate that the concolic executor in DigFuzz performs more executions but generates less inputs. Then, we can infer that the concolic executor in DigFuzz fails to solve some path constraints, which is possibly caused by the problem of constraint complexity.

5.7 Case Study

In this section, we demonstrate the effectiveness of our system by presenting an in-depth analysis via a case study. The binary used (*NRFIN_00017*) comes from the CQE dataset, which is also taken as a case study in the Driller paper [38].

Fig. 11 shows the core part of the source code for the binary. As depicted, the execution has to pass three nested checks (located in Ln. 4, 7, and 15) so as to trigger the vulnerability. We denote the three condition checks as *check_1*, *check_2*, and *check_3*.

5.7.1 Performance Comparison

Due to the three condition checks, AFL failed to crash the binary after running for 12 hours. In contrast, all of the hybrid fuzzing systems, DigFuzz, Random, QSYM, and Driller∗ were able to trigger the vulnerability.

Through examining the execution traces, we observed that the fuzzer in Driller∗ got stuck at the 57th second, the 95th second, and the 903rd second caused by *check_1*, *check_2*, and *check_3*, respectively. Per design, only at these moment will the concolic executor in Driller∗ be launched to help the fuzzer. Eventually, it took 2590 seconds for Driller∗ to generate a satisfying input for *check_3*, and guide the fuzzer to reach the vulnerable code.

Random, QSYM, and DigFuzz run the fuzzer and the concolic executor in parallel from the beginning. In each iteration, Random randomly selects an input for concolic execution, whereas QSYM selects inputs with the smallest size and DigFuzz selects the paths with the highest priority. Our examination shows that DigFuzz performed 7 concolic executions in 691 seconds, Random performed 26 executions in 1438 seconds, and QSYM performed 73 executions in 2859 seconds before they could generate an input satisfying *check_3*. Figs. 13 and 14 show the time stamps when concolic executions were invoked and how they helped the fuzzers in DigFuzz and Random. We did present the time stamps in QSYM because of the large number of performed concolic executions.

We can also observe that QSYM spent more time than Driller∗ in generating a satisfying input. Our manual analysis shows that the *NRFIN_00017* program subsequently receives inputs from *stdin* during the execution. Then, the

TABLE 8
Performance of Concolic Execution on Real-World Binaries

	Concolic executions				Imported inputs				Derived inputs			
	DigFuzz	Random	Driller∗	QSYM	DigFuzz	Random	Driller∗	QSYM	DigFuzz	Random	Driller∗	QSYM
libjpeg-9b+cjpeg	27	24	19	29	19	22	11	19	517	358	21	602
ncurses-6.1+infotocap	387	365	103	119	15	15	4	87	522	372	93	977
jhead-3.00	322	319	46	311	130	57	5	77	2197	1567	886	1206
nm-2.26.1	89	101	92	107	330	220	72	234	1026	963	365	1502
xpdf-4.00+pdfimages	173	177	87	186	45	53	6	64	1351	1190	147	520
libpng-1.6.36+pngfix	234	198	34	219	204	222	86	227	597	568	353	636
readelf-2.26.1	425	427	160	430	156	153	48	194	4900	4690	645	4772
size-2.26.1	29	29	28	29	258	180	128	204	2037	1673	784	1766
libtiff-4.0.10+tiffdump	333	339	0	365	6	2	**0**	4	952	939	**0**	906
libxpat-R.2.2.5+xmlwf	126	127	79	129	45	21	19	64	1233	1082	706	1168
objdump-2.26.1	194	106	104	135	9	6	3	12	238	148	111	124
exiv2-0.25	185	187	121	190	19	14	8	12	1625	692	67	2758
Total	2524	2399	873	2249	1236	965	390	1198	17195	14242	4178	16937

```
1  int main(void) {
     …
2    RECV(mode, sizeof(uint32_t));
3    switch (mode[0]) {
4      case MODE_BUILD: ret = do_build(); break;
     … }

5  int do_build() {
     …
6    switch(command) {
7      case B_CMD_ADD_BREAKER:
8      model_id = recv_uint32();
9      add_breaker_to_load_center(model_id, &result);
10     break;
     …}}

11 int8_t add_breaker_to_load_center() {
12       get_new_breaker_by_model_id(…);}

13 int8_t get_new_breaker_by_model_id(…) {
14       switch(model_id){
15         case FIFTEEN_AMP:
16           //vulnerability
17           break;
     …}}
```

Fig. 11. The vulnerability in *NRFIN_00017*.

size of input will increase along with the execution to bypass *check_1*, *check_2*, and *check_3*. As QSYM selects inputs with the decreasing order of input sizes, it cannot select a suitable input to reach *check_3* until it finished executing all smaller inputs. That is why QSYM performed the largest number of concolic executions and spent longest time before triggering the vulnerability.

In terms of the input generation, DigFuzz managed to generate 96 inputs within which 37 were imported by fuzzing. Random generated 373 inputs and 44 of them were imported. QSYM generated 2750 inputs and 31 of them were imported. Moreover, by the time of the 691st second, DigFuzz generated 37 imported inputs while both Random and QSYM can only generate 4 inputs. These numbers show that DigFuzz could generate inputs with much higher quality than Random and QSYM.

5.7.2 In-Depth Analysis

We further present the details on how concolic execution helped the fuzzer to bypass these three checks in DigFuzz. Fig. 12 briefly shows the execution tree for *NRFIN_00017*, in which the path that leads to the vulnerability is marked as red. To trigger the vulnerability, the execution has to go through three checks (*check_1*, *check_2* and *check_3*) and dives into three functions (*do_build()*, *add_breaker_to_load_center()* and *get_new_breaker_by_model_id()*).

The fuzzer was blocked at *check_1* and got stuck quickly after start. For this specific branch, all DigFuzz, Random, QSYM, and Driller∗ quickly generated a satisfying input to bypass *check_1*. After this, the fuzzers went into *do_build()*, quickly generated 23 interesting inputs in less than 1 minute and then reached *check_2*. From Fig. 13, we can observe from that the concolic executor in DigFuzz accurately selected the one that corresponded to *check_2* from the 23 inputs, and solved the condition in just one run at the 636th second. Further, the fuzzer went into the third function *get_new_breaker_by_model_id()* and reached *check_3*. Note that even though the fuzzer was blocked by *check_3*, it did not get stuck, because there are a number of paths that the fuzzer can go through as shown in Fig. 12. At this moment, the concolic executor was handed with 97 inputs from the fuzzer and had to pick the right one to reach *check_3*. Fig. 13 shows that DigFuzz took only 2 concolic executions to bypass *check_3*. Eventually, DigFuzz generated an input satisfying *check_3* at the 691st second.

As a comparison, we examined how the concolic executors work in Random, QSYM, and Driller∗. When the fuzzer had bypassed *check_1*, it quickly discovered more blocks thus retained amount of inputs. As shown in Fig. 12, there are a number of paths that the fuzzer can go through. Therefore, the fuzzer took a long time to get stuck again. However, along with the red path in Fig. 12, the fuzzer quickly got blocked at *check_3*. Driller∗ will not identify this specific branch until the the fuzzer gets stuck again. QSYM requires to go through all the smaller inputs. Random selects inputs without recognizing their corresponding execution paths. With path prioritization, DigFuzz is able to identify specific paths that block the fuzzer in time.

By monitoring the status of the fuzzer, we also observe that the fuzzer got stuck for 8 times in Driller∗, 4 times in QSYM, 3 times in Random, and only 1 time in DigFuzz. This result indicates that the path prioritization in DigFuzz was

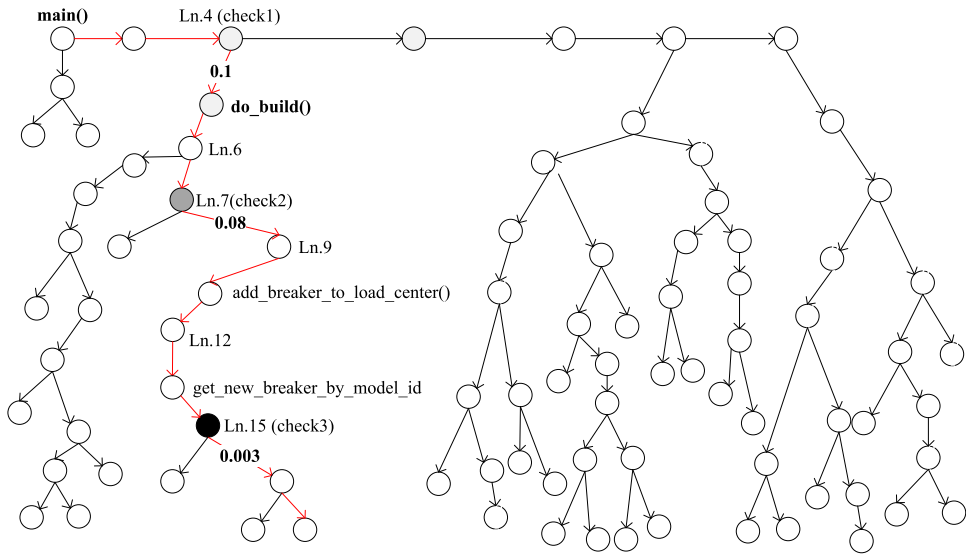


Fig. 12. The execution tree for *NRFIN_00017*.

able to generate satisfying inputs for specific paths that block the fuzzer in time. As a result, the fuzzer avoided being stuck for majority of checks.

6 Discussion

6.1 Threats to Validity

We adopt two concolic executors, *angr* [37] and *QSYM* [46], to build our prototype. Specifically, we utilize *QSYM* [46] to evaluate real-world programs, because *angr* does not have sufficient support for real-world programs [31]. However, we still encountered the incompatibility problem that *QSYM* [46] cannot work on some real-world programs, and then we excluded real-world programs on which *QSYM* [46] cannot work. With this impact, the results may not be fully representative of real-world programs.

6.2 Limitations

Our approach regards the test cases generated by a fuzzer as samples for estimating probabilities with the Monte Carlo method, which expects uniformly random samples for the most accurate estimation. However, grey-box fuzzers (such AFL adopted in DigFuzz) are not truly uniformly randomized, in terms of seed mutation and program space exploration. Hence, calculated probabilities could be biased, and we may miss certain paths that are difficult for AFL to explore if the paths are under-sampled. Nonetheless, we argue that this bias does not hinder our ultimate goal for directing symbolic execution to aid AFL by identifying difficult paths. To achieve this goal, the probability estimation is meant to be designed in such a way that it reflects the difficulty for the fuzzer (AFL in our case) to explore certain paths during testing. The estimated difficulty of one path is calculated from the runtime working states of AFL, which is specific for each fuzz testing, rather than an universal difficulty. Besides, the probability estimation iteratively evolves with the increased number of samples. Therefore, our approach can identify difficult paths that block AFL at the current state of fuzz testing.

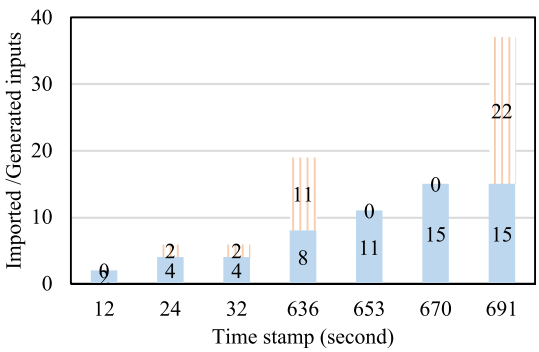


Fig. 13. Concolic executions by DigFuzz on *NRFIN_00017*.

Second, although the "discriminative dispatch" in DigFuzz is designed to be a lightweight approach, it still imposes some runtime and memory consumption overhead including collecting runtime information of fuzzing and constructing the execution tree. The runtime overhead is 4 percent on average (596 executions per second in the pure fuzzing versus 582 executions per second in DigFuzz). The total memory consumption of the execution tree varies from programs, which ranges from 10M to 100M.

Third, since DigFuzz only estimates the difficulty of exploring paths for a fuzzer but does not consider the complexity of concolic execution, it is possible that the constraints for the picked path are unsolvable, which may result in a waste of concolic execution cycle. We consider solving them as future work.

7 Related Work

Fuzzing and symbolic execution are the two mainstream techniques for program testing. Many prior efforts have been made to improve them [2], [3], [11], [14], [18], [19], [22], [26], [28], [32], [33], [35], [39], [45]. CollAFL [18] is a coverage sensitive fuzzing solution, which mitigates path collisions by providing more accurate coverage information. It also utilizes the coverage information to apply three new fuzzing strategies. Intriguer [14] optimizes symbolic execution with field-level knowledge. It performs instruction-level

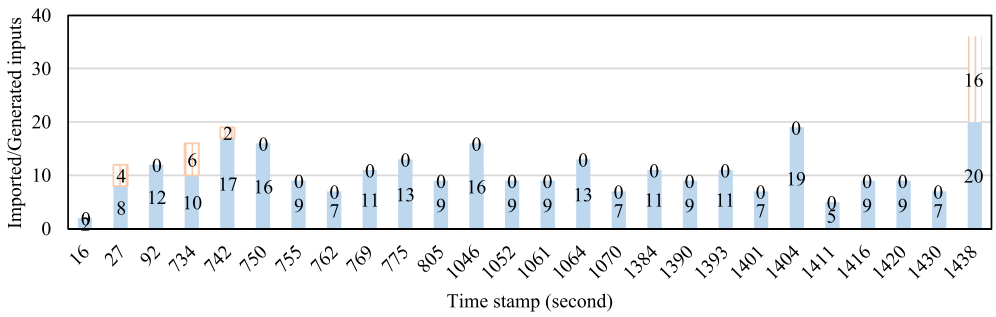


Fig. 14. Concolic executions by Random on *NRFIN_00017*.

taint analysis and then reduces the execution traces for tainted instructions that accessed a wide range of input bytes, and infers input fields to build field transition trees. With these optimizations, Intriguer can efficiently perform symbolic emulation for more relevant instructions and invoke a solver for complicated constraints only. The main contribution of DigFuzz is to propose a more effective strategy to combine fuzzing with concolic execution. The advances of fuzzing and concolic execution are out of our scope.

*Hybrid Fuzzing System.*Most hybrid fuzzing systems follow the observation to augment fuzzing with selective symbolic execution [9], [27], [38], [40]. TaintScope [40] deploys dynamic taint analysis to identify the checksum checkpoints and then applies symbolic execution to generate inputs satisfying checksum. T-Fuzz [31] first allows a fuzzer to work on the transformed program by removing sanity checks, and then leverages a symbolic execution-based approach to filter out false positives. SAVIOR [13] proposes a bug-driven hybrid fuzzing system. It prioritizes the seeds that have higher potential leading to the discovery of more bugs for concolic execution. Besides, it enables security checks during concolic execution. HFL [25] combines fuzzing with symbolic execution for addressing kernel-specific fuzzing challenges. PANGOLIN [23] designs incremental hybrid fuzzing with polyhedral path abstraction, which preserves the exploration state in the concolic execution stage and allows more effective mutation and constraint solving over existing techniques. Compared with these techniques, DigFuzz prioritizes paths by quantifying the difficulties for fuzzing to explore a path based on coverage statistics.

Another type of hybrid fuzzing system is to regard the symbolic execution as a guider for input generation or path selection. Pak [30] proposes a hybrid fuzzing system to apply symbolic execution to collects path constraints, then the system generates inputs that respect the path predicates and transits to the fuzzer. DeepFuzz [5] applies probabilistic symbolic execution to assign probabilities to program paths, and then takes these probabilities to guide the path exploration in fuzzing.

*Symbolic Execution.*Path prioritization is promising for mitigating the path explosion problem in dynamic symbolic execution [6], [7], [8]. These heuristics include using the control-flow graph to guide the exploration, frequency-based and random-based techniques [6], [7]. Path prioritization is adopted to combine with evolutionary search, in which a fitness function is defined to guide the symbolic execution [1]. Compared with these path exploration techniques, the path prioritization in DigFuzz is to prioritize paths with probabilities how difficult for fuzzing to pass through.

Directed symbolic execution also employs path prioritization to reach a target. These techniques aim to search for a feasible path for a target statement or branch [36], [44]. Compared with directed symbolic execution techniques, the path prioritization in DigFuzz is to identify the targeted paths for concolic execution, instead of searching for a feasible path for a given target.

*Seed Scheduling in Fuzzing.*Seed selection plays an important role in fuzzing, and several studies have been proposed to improve the seed scheduler [4], [10], [43] by prioritizing seed inputs. The basic insight behind these seed scheduling technique is to search for a seed on which the mutated execution is more likely to discover new program states. In our future work, we plan to design scheduling techniques to offload the fuzzer with paths that are difficult to explore.

8 Conclusion

We perform a thorough investigation on some state-of-the-art hybrid fuzzing systems and point out several fundamental limitations in the "optimal concolic testing" and "demand launch" strategies deployed in these systems. We further design a "discriminative dispatch" strategy and propose a probabilistic hybrid fuzzing system to better utilize the capability of concolic execution. We implement a prototype system DigFuzz based on the design and conduct a comprehensive evaluation using two popular datasets and real-world programs. The evaluation results show that the concolic execution in DigFuzz contributes much more to the increased code coverage and increased number of discovered vulnerabilities compared with state-of-the-art hybrid fuzzing systems.

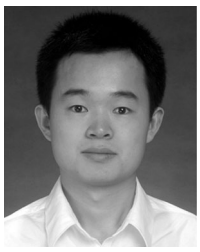
Acknowledgments

This work was supported in part by the National Natural Science Foundation of China under Grant No. 61672394, U1836112, and 61872273, and in part by the Fundamental Research Funds for the Central Universities.

References

[1] A. Baars *et al.*, "Symbolic search-based testing," in *Proc. 26th Int. Conf. Autom. Softw. Eng.*, 2011, pp. 53–62.
[2] T. Blazytko *et al.*, "GRIMOIRE Synthesizing structure while fuzzing," in *Proc. 28th USENIX Secur. Symp*, 2019, pp. 1985–2002.
[3] M. Bohme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed Greybox fuzzing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 2329–2344.
[4] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based Greybox fuzzing as Markov chain," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 1032–1043.

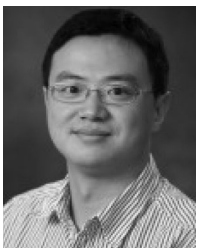
[5] K. Böttinger and C. Eckert, "DeepFuzz: Triggering vulnerabilities deeply hidden in binaries," in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Berlin, Germany: Springer, 2016, pp. 25–34.
[6] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. 8th USENIX Conf. Operating Syst. Des. Implementation*, 2008, pp. 209–224.
[7] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler, "EXE: automatically generating inputs of death," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 2, pp. 1–38, 2008.
[8] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, 2013.
[9] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proc. IEEE Symp. Secur. Privacy*, 2012, pp. 380–394.
[10] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *Proc. IEEE Symp. Secur. Privacy*, 2015, pp. 725–741.
[11] P. Chen and C. Hao, "Angora: Efficient fuzzing by principled search," in *Proc. 39th IEEE Symp. Secur. Privacy*, 2018, pp. 711–725.
[12] P. Chen, J. Liu, and H. Chen, "Matryoshka: fuzzing deeply nested branches," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 499–513.
[13] Y. Chen *et al.*, "SAVIOR: Towards bug-driven hybrid testing," in *Proc. 41st IEEE Symp. Secur. Privacy*, 2020, pp. 1580–1596.
[14] M. Cho, S. Kim, and T. Kwon, "Intriguer: Field-level constraint solving for hybrid fuzzing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 515–530.
[15] DARPA, "Cyber grand challenge challenge repository," 2017. [Online]. Available: http://www.lungetech.com/cgc-corpus/
[16] B. Dolan-Gavitt, "Of bugs and baselines," 2018. [Online]. Available: http://moyix.blogspot.com/2018/03/of-bugs-and-baselines.html
[17] B. Dolan-Gavitt *et al.*, "LAVA: Large-scale automated vulnerability addition," in *Proc. IEEE Symp. Secur. Privacy*, 2016, pp. 110–121.
[18] S. Gan *et al.*, "CollaFL: Path sensitive fuzzing," in *Proc. 39th IEEE Symp. Secur. Privacy*, 2018, pp. 679–696.
[19] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proc. 31st Int. Conf. Softw. Eng.*, 2009, pp. 474–484.
[20] J. Geldenhuys, M. B. Dwyer, and W. Visser, "Probabilistic symbolic execution," in *Proc. Int. Symp. Softw. Testing Anal.*, 2012, pp. 166–176.
[21] W. R. Gilks, "Markov chain Monte Carlo," *Encyclopedia of Biostatistics*, 2005.
[22] I. Haller, A. Slowinska, H. Bos, and M. M. Neugschwandtner, "Dowsing for overflows: A guided fuzzer to find buffer boundary violation," in *Proc. 22nd USENIX Conf. Secur.*, 2013, pp. 49–64.
[23] H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang, "Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction," in *Proc. 41st IEEE Symp. Secur. Privacy*, 2020, pp. 1613–1627.
[24] X. Jia, C. Zhang, P. Su, Y. Yang, H. Huang, and D. Feng, "Towards efficient heap overflow discovery," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 989–1006.
[25] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "HFL: Hybrid fuzzing on the linux kernel," in *Proc. 27th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2020, pp. 1–17.
[26] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: program-state based binary fuzzing," in *Proc. 11th Joint Meet. Foundations Softw. Eng.*, 2017, pp. 627–637.
[27] R. Majumdar and K. Sen, "Hybrid concolic testing," in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 416–426.
[28] V. J. M. Manès *et al.*, "The art, science, and engineering of fuzzing: A survey," *IEEE Trans. Softw. Eng.*, to be published, doi: 10.1109/TSE.2019.2946563.
[29] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, 1990.
[30] B. S. Pak, "Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution," Master's thesis, School of Computer Science Carnegie Mellon University, 2012.
[31] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-Fuzz: Fuzzing by program transformation," in *Proc. 39th IEEE Symp. Secur. Privacy*, 2018, pp. 697–710.
[32] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "SlowFuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 2155–2168.
[33] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware evolutionary fuzzing," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017, pp. 1–14.
[34] C. P. Robert, *Monte Carlo Methods*. Hoboken, NJ, USA: Wiley, 2004.
[35] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-assisted feedback fuzzing for OS kernels," in *Proc. 26th USENIX Conf. Secur. Symp.*, 2017, pp. 167–182.
[36] H. Seo and S. Kim, "How we get there: A context-guided search strategy in concolic testing," in *Proc. 22nd ACM SIGSOFT Int. Symp. Foundations Softw. Eng.*, 2014, pp. 413–424.
[37] Y. Shoshitaishvili *et al.*, "SOK:(state of) the art of war: Offensive techniques in binary analysis," in *Proc. IEEE Symp. Secur. Privacy*, 2016, pp. 138–157.
[38] N. Stephens *et al.*, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2016, pp. 1–16.
[39] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *Proc. 38th IEEE Symp. Secur. Privacy*, 2017, pp. 579–594.
[40] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Proc. IEEE Symp. Secur. Privacy*, 2010, pp. 497–512.
[41] X. Wang, J. Sun, Z. Chen, P. Zhang, J. Wang, and Y. Lin, "Towards optimal concolic testing," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 291–302.
[42] Wikipedia, "Rule of three (statistics)," 2017. [Online]. Available: https://en.wikipedia.org/wiki/Rule_of_three_%28statistics
[43] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2013, pp. 511–522.
[44] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw..*, 2009, pp. 359–368.
[45] T. Yue *et al.*, "EcoFuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit," in *Proc. 29th USENIX Secur. Symp.*, 2020, pp. 2307–2324.
[46] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 745–761.
[47] M. Zalewski, "American Fuzzy Lop," 2017. [Online]. Available: http://lcamtuf.coredump.cx/afl/
[48] L. Zhao, Y. Duan, H. Yin, and J. Xuan, "Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2019, pp. 1–15.



Lei Zhao received the BSc and PhD degrees in computer science and engineering, both from Wuhan University, China, in 2007 and 2012, respectively. He is an associate professor with the School of Cyber Science and Engineering, Wuhan University, China. His research interests include software testing and debugging, program analysis, and software security.

Peng Cheng received the BSc degree in computer science and engineering from Wuhan University, China, in 2018. He is currently working toward the graduate degree with the School of Cyber Science and Engineering, Wuhan University, China. His research interests include software testing and concolic execution.

Yue Duan received the PhD degree in computer science from UC Riverside, in 2019. He had his postdoctoral training at Cornell University and the University of Utah. He is an assistant professor with the Department of Computer Science at the Illinois Institute of Technology. His research interests include program analysis and software security.



Heng Yin is a professor with the Department of Computer Science and Engineering at UC Riverside. Before joining UC Riverside, he was with Syracuse University, from September 2009 to June 2016, as assistant professor and then associate professor. His research interests include detection and analysis, vulnerability discovery, program hardening, and digital forensics.

Jifeng Xuan (Member, IEEE) received the BSc and PhD degrees from the Dalian University of Technology, China. He is a professor with the School of Computer Science, Wuhan University, China. He was previously a postdoctoral researcher at INRIA Lille Nord Europe, France. His research interests include software testing and debugging, software data analysis, and search-based software engineering. He is a member of ACM and CCF.