Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing

Lei Zhao* Yue Duan† Heng Yin† Jifeng Xuan‡
*School of Cyber Science and Engineering, Wuhan University, China
*Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, China
†University of California, Riverside
‡School of Computer Science, Wuhan University, China
leizhao@whu.edu.cn {yduan005, heng}@cs.ucr.edu jxuan@whu.edu.cn

***Abstract*—Hybrid fuzzing which 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, the state-of-the-art hybrid fuzzing systems deploy "demand launch" and "optimal switch" strategies. Although these ideas sound intriguing, we point out several fundamental limitations in them, due to oversimplified assumptions. We then propose a novel "discriminative dispatch" strategy to better utilize the capability of concolic execution. We design a novel Monte Carlo based probabilistic path prioritization model to quantify each path's difficulty and prioritize them for concolic execution. This model treats fuzzing as a random sampling process. It calculates each path's probability based on the sampling information. Finally, our model prioritizes and assigns the most difficult paths to concolic execution. We implement a prototype system DigFuzz and evaluate our system with two representative datasets. 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 vs. 5) and producing more code coverage (18.9% vs. 3.8%) on the CQE dataset than the concolic execution in Driller.**

I. Introduction

Software vulnerability is considered one of the most serious threats to the cyberspace. As a result, it is crucial to discover vulnerabilities in a piece of software [12], [16], [25], [27], [32]. Recently, hybrid fuzzing, a combination of fuzzing and concolic execution, has become increasingly popular in vulnerability discovery [5], [29], [31], [39], [42], [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 incapable of exploring paths containing *specific* branches (branches that have very narrow satisfying value spaces) [27]. In contrast, concolic execution is able to generate concrete inputs that ensure the program to execute along a specific execution path, but it suffers from the path explosion problem [9]. 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 [39] and hybrid concolic testing [29] take a "demand launch" strategy: fuzzing starts first and concolic execution is launched only when the fuzzing cannot make any progress for a certain period of time, a.k.a., stuck. A recent work [42] proposes an "optimal switch" strategy: it quantifies the costs for exploring each path by fuzzing and concolic execution respectively and chooses the more economic method for exploring that path.

We have evaluated both "demand launch" and "optimal switch" strategies using the DARPA CQE dataset [13] and LAVA-m dataset [15], 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 in stuck state. 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 then overwhelmed by this massive number of missed paths, and might generate a helping input for a specific path after a long time. By then, the fuzzer might have already generated a good input to traverse that specific path, rendering the whole concolic execution useless.

Likewise, although the "optimal switch" strategy aims to make optimal decisions based on a solid mathematical model (i.e., Markov Decision Processes with Costs, MDPC for short),

The main work was conducted when Lei Zhao worked at University of California Riverside as a Visiting Scholar under Prof. Heng Yin's supervision.

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23504
www.ndss-symposium.org

it is nontrivial to quantify the costs of fuzzing and concolic execution for each path. For instance, to quantify the cost of concolic execution for a certain path, MDPC requires to collect the path constraint, which is already expensive. As a result, the overall throughput of MDPC is greatly reduced. Furthermore, when quantifying the cost of fuzzing, MDPC assumes a uniform distribution on all test cases. This assumption is oversimplified, as many state-of-the-art fuzzing techniques [4], [12], [16] are adaptive and evolutionary. 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 propose a "discriminative dispatch" strategy to better combine fuzzing and concolic execution. That is, we prioritize paths so that concolic execution *only* works on selective paths that are *most* difficult for fuzzing to break through. Therefore, the capability of concolic execution is better utilized. Then the key for this "discriminative dispatch" strategy to work is a lightweight method to quantify the difficulty level for each path. Prior work solves this problem by performing expensive symbolic execution [18], and thus is not suitable for our purpose.

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. To be more specific, we quantify a path's difficulty by its probability of how likely a random input can traverse this path. To calculate this probability, we use the Monte Carlo method [35]. The core idea is to treat fuzzing as a random sampling process, consider random executions as samples to the whole program space, and then calculate each path's probability based on the sampling information.

We have implemented a prototype system called DigFuzz. It leverages a popular fuzzer, American Fuzzy Lop (AFL) [47], as the fuzzing component, and builds the concolic executor on top of Angr, an open-source symbolic execution engine [38]. We evaluate the effectiveness of DigFuzz using the CQE binaries from DARPA Cyber Grand Challenge [13] and the LAVA dataset [15]. The evaluation results show that the concolic execution in DigFuzz contributes significantly more to the increased code coverage and increased number of discovered vulnerabilities than state-of-the-art hybrid systems. To be more specific, the concolic execution in DigFuzz contributes to discovering more vulnerabilities (12 vs. 5) and producing more code coverage (18.9% vs. 3.8%) on the CQE dataset than the concolic execution in Driller [39].

Contributions. The contributions of the paper are summarized as follows:

- We conduct an independent evaluation of two state-of-the-art hybrid fuzzing strategies ("demand launch" and "optimal switch"), 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 and LAVA dataset. Our experiments demonstrate that DigFuzz outperforms the state-of-the-art hybrid systems Driller and MDPC with respect to more discovered vulnerabilities and higher code coverage.

II. Background and Motivation

Fuzzing [30] and concolic execution [9] are two representative techniques for software testing and vulnerability detection. With the observation that fuzzing and concolic execution can complement each other in nature, a series of techniques [5], [29], [31], [39], [42] have been proposed to combine them together and create hybrid fuzzing systems. In general, these hybrid fuzzing systems fall into two categories: "demand launch" and "optimal switch".

A. Demand Launch

The state-of-the-art hybrid schemes such as Driller [39] and hybrid concolic testing [29] deploy a "demand launch" strategy. In Driller [39], 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 [29] obtains both a deep and a 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 nutshall, 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 discovery new coverage.

Observations. To assess the performance of the "demand launch" strategy, we carefully examine how Driller works on 118 binaries from DARPA Cyber Grand Challenge (CGC) for 12 hours and find five interesting yet surprising facts.

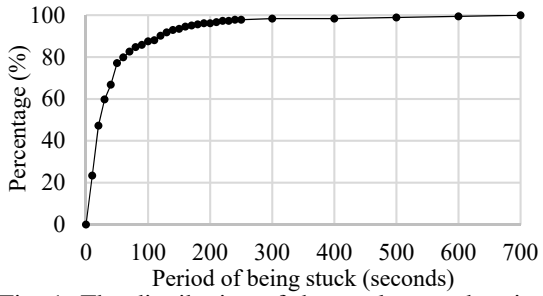


Fig. 1: The distribution of the stuck state durations

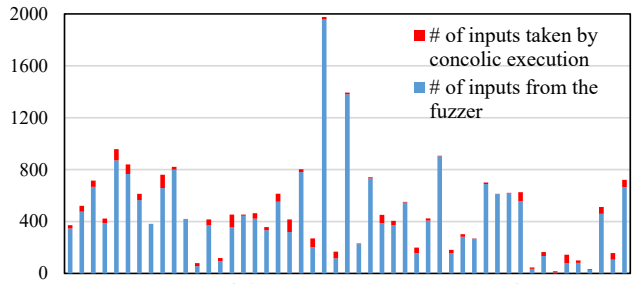


Fig. 2: The number of inputs retained by the fuzzer and the number of inputs taken by concolic execution.

(1) Driller invoked concolic execution on only 49 out of 118 binaries, which means that the fuzzer only gets stuck on these 49 binaries. This fact is on par with the numbers (42) reported in the paper of Driller [40].
(2) For the 49 binaries from Fact 1, we statistically calculate the stuck time periods, and the the distribution of stuck time periods is shown in Figure 1. We can observe that that more than 85% of the stuck time periods are under 100 seconds.
(3) On average, it takes 1654 seconds for the concolic executor to finish the dynamic symbolic execution on one concrete input.
(4) Only 7.1% (1709 out of 23915) of the inputs retained by the fuzzer are processed by the concolic executor within the 12 hours of testing. Figure 2 presents this huge gap between the number of inputs taken by concolic execution and that the number of inputs retained by fuzzing.
(5) The fuzzer in Driller can indeed get help from concolic execution (import at least one input generated by concolic execution) on only 13 binaries among the 49 binaries from Fact 1, with a total of 51 inputs imported after 1709 runs of concolic execution.

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 Fact 1, 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 fuzzing. Further combining with Fact 2, 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% of the stuck states are under 100 seconds). These facts break the Assumption 1 described above.

Second, the "demand launch" strategy can not 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 (Fact 3). On the other hand, a fuzzer often retains much more inputs than what concolic execution could handle (Fact 4). As a result, the input corresponding to the specific branch that block 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, the Assumption 2 described above does not really hold in practice. This conclusion can be further confirmed by Fact 5 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.

B. Optimal Switch

The "optimal switch" strategy aims to make an optimal decision on which method to use to explore a given execution path, based on a mathematical model (i.e., Markov Decision Processes with Costs, MDPC for short). To achieve optimal performance, MDPC always selects the method with lower cost to explore each path. In order for this strategy to work well, the following assumptions must hold:

(1) The costs for exploring a path by fuzzing and concolic execution can be accurately estimated.
(2) The overhead of cost estimation is negligible.
(3) The algorithm for making optimal decisions is lightweight.

Observations. To assess the performance of "optimal switch", we evaluate how MDPC works on the 118 binaries from the CQE dataset for 12 hours and have 3 interesting observations.

TABLE I: Execution Time Comparison

	Fuzzing	Concolic execution	MDPC decision
Minimum	0.0007s	18s	0.16s
25% percentile	0.0013s	767s	13s
Median	0.0019s	1777s	65s
Average	0.0024s	1790s	149s
75% percentile	0.0056s	2769s	213s
Maximum	0.5000s	3600s	672s

(1) Table I shows the throughput gap among fuzzing, concolic execution, and the optimal decision in MDPC. We can observe that the optimal decisions is very expensive, which is several thousand times larger than fuzzing.
(2) As MDPC makes optimal decision before exploring each path, the overall analysis throughput is significantly reduced, from 417 executions per second in pure fuzzing to 2.6 executions per second in MDPC.
(3) With the impact of reduced throughput, MDPC discovers vulnerabilities only in 29 binaries, whereas the pure fuzzing can discover vulnerabilities in 67 binaries.

As MDPC makes the optimal decision before exploring each path, the expensive optimal decision takes away the advance of the high throughput of fuzzing. As an optimization, we can move the optimal decision out, make it work in parallel

with fuzzing, and build a concurrent MDPC. That is, the optimal decision in the concurrent MDPC does not interfere the working progress of fuzzing, and it just collected coverage statistics from fuzzing to calculate cost. From the evaluation of this concurrent MDPC, we have another observation.

(4) 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 several 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 [42]) to every path constraint.

Limitations. The aforementioned observations indicate that the key limitation of the "optimal switch" strategy is that estimating the cost for exploring a path by fuzzing and concolic execution is heavyweight and inaccurate, which overshadows the benefit of making optimal solutions.

First, estimating the cost of concolic execution relies on collecting path constraints and identifying the solving cost for these constraints. As collecting path constraints requires converting program statements into symbolic expressions, such interpretation is also heavyweight, especially for program with long paths. In addition, MDPC designs a greedy algorithm for optimal decision. This algorithm depends on path-sensitive program analysis. For programs with large states, the path-sensitive analysis is also heavyweight.

Second, it is nontrivial to accurately estimate cost for exploring a given path by fuzzing and concolic execution. MDPC estimates the solving cost based on the complexity of equalities, and estimates the cost of random testing based on coverage statistics. These estimations are concerned with the runtime throughput of fuzzing, the performance of the constraint solver, and the symbolic execution engine, and each of them are different program analysis techniques in nature. Therefore, it is challenging to define a unified metric to evaluating the cost of different techniques.

III. 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.

A. 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 [45] and probabilistic symbolic execution [5]. However, these techniques do not solve our problem: if we have already performed heavyweight analysis to quantify the difficulty of a path, we might as well just solve the path constraints and generate an input to traverse the path. A recent study [42] proposes a theoretical framework for optimal concolic testing. It defines the optimal strategy based on the probability of program paths and the cost of constraint solving, and then reduces the problem as Markov Decision Processes with Costs (MDPC for short). This study shares the similar problem scope with our work. However, the Markov Decision Process itself is heavyweight for programs with large state space. Furthermore, the costs of fuzzing and concolic execution are challenging to evaluate and normalize for comparison.

B. 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 challenges. 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, 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.

According to the Monte Carlo method, we can simply estimate the probability of a path by statistically calculating the ratio of executions traverse 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.

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

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 [43] 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% 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 is larger than 30.

Probability for each path. To calculate the probability for a path, we apply the Markov Chain model [19] 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.

$$P\,(path_j) = \prod\{P\,(br_i)\,|br_i \in path_j\} \tag{2}$$

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 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.

C. 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*, α), 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.

IV. Design and Implementation

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

A. System Overview

Figure 3 shows the 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 $angr$ [38], an open-source symbolic execution engine, the same as Driller [39].

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 identifies 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 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 by solving path constraints, 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 missed branches along the trace, 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.

B. Execution Sampling

Random sampling is required for DigFuzz to calculate probabilities using the Monte Carlo method [35]. 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 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* [43] where the interval from 0 to 3/n is a 95%

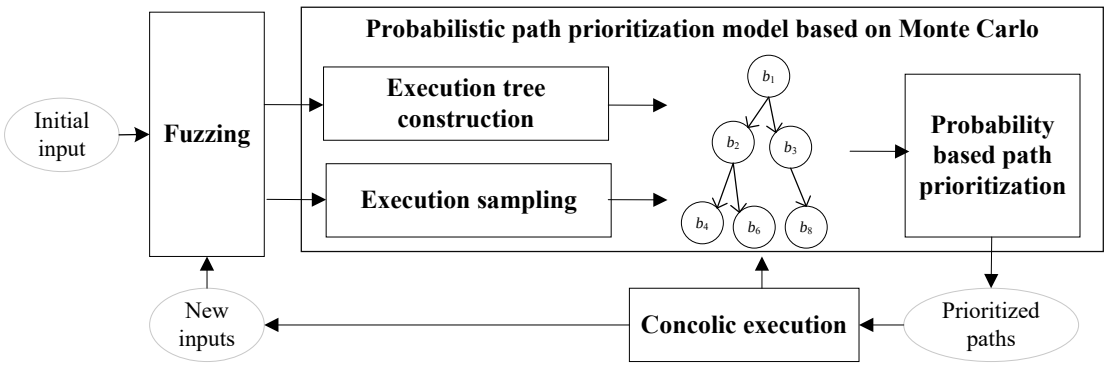


Fig. 3: Overview of DigFuzz

Algorithm 1 Execution Sampling

1: $P \leftarrow$ {Target binary}
2: $Fuzzer \leftarrow$ {Fuzzer in DigFuzz }
3: $Set_{inputs} \leftarrow$ {Initial seeds}
4: $HashMap_{CovStat} \leftarrow \emptyset; Set_{NewInputs} \leftarrow \emptyset$
5:
6: **while** True **do**
7: $\quad Set_{NewInputs} \leftarrow Fuzzer\{P, Set_{inputs}\}$
8: $\quad$ **for** $input \in Set_{NewInputs}$ **do**
9: $\quad\quad Coverage \leftarrow$ GetCoverage$(P, input)$
10: $\quad\quad$ **for** $branch \in Coverage$ **do**
11: $\quad\quad\quad Index \leftarrow$ Hash$(branch)$
12: $\quad\quad\quad HashMap_{CovStat}\{Index\}++$
13: $\quad\quad$ **end for**
14: $\quad$ **end for**
15: $\quad Set_{inputs} \leftarrow Set_{inputs} \cup Set_{NewInputs}$
16: **end while**
output the $HashMap_{CovStat}$ as coverage statistics

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 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.

C. Execution Tree Construction

As depicted in Figure 3, DigFuzz generates the MCP^3 based execution tree using the runtime information from the 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}

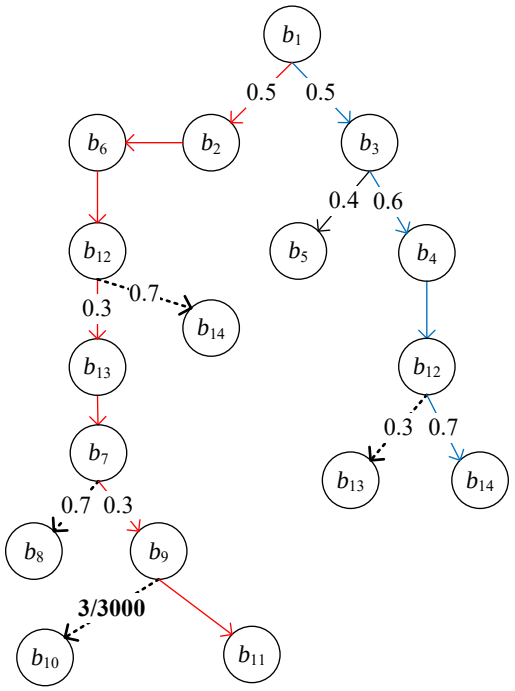
	void main(argv) {		**int** chk_in () {
	recv(in);	b_6	res = is_valid(in)
	switch (argv[1]) {	b_7	if (!res)
b_1	case 'A':	b_8	return;
b_2	chk_in(in);	b_9	if (strcmp(in, 'BK') == 0);
	break;	b_{10}	//**vulnerability**
b_3	case 'B':	b_{11}	else … }
b_4	is_valid(in);		**int is_valid**(in) {
	break;	b_{12}	if all_char(in)
b_5	default: …	b_{13}	return 1;
	}}	b_{14}	return 0; }

Fig. 4: Running Example

Algorithm 2 Execution Tree Construction

1: $CFG \leftarrow$ {Control flow graph for the target binary.}
2: $Set_{inputs} \leftarrow$ {Inputs retained by the fuzzer}
3: $HashMap_{CovStat} \leftarrow$ {Output from Algorithm 1}
4: $ExecTree \leftarrow \emptyset$
5:
6: **for** $input \in Set_{inputs}$ **do**
7: $\quad trace \leftarrow$ TraceAnalysis$(input)$
8: $\quad$ **if** $trace \notin ExecTree$ **then**
9: $\quad\quad ExecTree \leftarrow ExecTree \cup trace$
10: $\quad$ **end if**
11: **end for**
12: **for** $br_i \in ExecTree$ **do**
13: $\quad br_j \leftarrow$ GetNeighbor(br_i, CFG)
14: $\quad prob \leftarrow$ CalBranchProb$(br_i, br_j, HashMap_{CovStat})$
15: $\quad$ LabelProb$(ExecTree, br_i, prob)$
16: **end for**
output $ExecTree$ as the MCP^3 based execution tree

and the coverage statistics $HashMap_{CovStat}$, which is also 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



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>, $P(P_2)$=0.063
P_3=<b_1, b_2, b_6, b_{12}, b_{14}>), $P(P_3)$=0.09
P_4=<b_1, b_3, b_4, b_{12}, b_{13}>, $P(P_4)$=0.105

Fig. 5: The execution tree with probabilities

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 the execution tree $ExecTree$ with the calculated probabilities (Ln. 15) and outputs the newly labeled $ExecTree$.

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. Traces on these retained seeds is a promising approach to model the program state explored by fuzzing. 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 conjuncted 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 [13], and the code piece is shown in Figure 4. The vulnerability is guarded by a specific string, which is hard for fuzzing to detect. Figure 5 illustrates the MCP^3 based execution tree for the running example in Figure 4. Each node represents a basic block. Each edge refers 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.

Algorithm 3 Path Prioritization in DigFuzz

```
1:  ExecTree ← {Output from Algorithm 2}
2:  Set_Prob ← ∅
3:  for trace ∈ ExecTree do
4:      for br_i ∈ trace do
5:          br_j ← GetNeighbor(br_i, CFG)
6:          missed ← GetMissedPath(trace, br_i, br_j)
7:          if missed ∉ ExecTree then
8:              prob ← CalPathProb(missed)
9:              Set_Prob ← {trace, missed, prob}
10:         end if
11:     end for
12: end for
    output Set_Prob as missed paths with probabilities corresponding to each trace
```

D. 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.

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. DigFuzz 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 Figure 4 as an example. In Figure 5, 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 P_1, DigFuzz 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.

V. Evaluation

In this section, we conduct comprehensive evaluation on the effectiveness and the runtime performance of DigFuzz by comparing with the state-of-the-art hybrid fuzzing systems, Driller [39] and MDPC [42], with respect to code coverage, number of vulnerabilities discovered, and contribution of concolic execution. In the end, we conduct a detailed analysis of DigFuzz using a case study.

A. Datasets

We leverage two datasets: the DARPA CGC Qualifying Event (CQE) dataset [13], the same as Driller [39], and the LAVA dataset, a widely adopted dataset in recent studies [12], [27], [34]. Both of them provide ground-truth for verifying detected vulnerabilities.

The CQE dataset contains 131 applications which are deliberately designed by security experts to test automated vulnerability detection systems. Every binary is injected 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 and use 126 binaries as in Driller.

For the LAVA dataset [15], we adopt `LAVA-M` as previous techniques [12], [27], 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.

B. 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 [16], Angora [12], AFLfast [4] and VUzzer [34].

To quantify the contribution of concolic execution, we leverage unaided 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 systems, Driller and MDPC.

We use Driller[1] to represent this configuration. Moreover, to evaluate the impact of 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 II-A.

[1]The configuration of Driller in our work is different from Driller paper as further discussed in Section V-C

In the original MDPC model [42], 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 contibues 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 challenge problem, we simply assign every path constraints with the solving cost of 50, which is the highest solving cost as defined [42]. Please note that with this configuration, the time cost by MDPC for optimal decision is lower, because it does not spent effort in collecting and estimating path constraints.

C. Experiment setup

The original Driller [39] adopts a shared pool design, where the concolic execution pool is shared among all the fuzzer instances. With this design, when fuzzing 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 details, we allocates 2 fuzzing instances for testing binaries with AFL, 2 fuzzing instances and 1 concolic execution instance with Driller, Random, 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 is configured with 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 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.

D. 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.

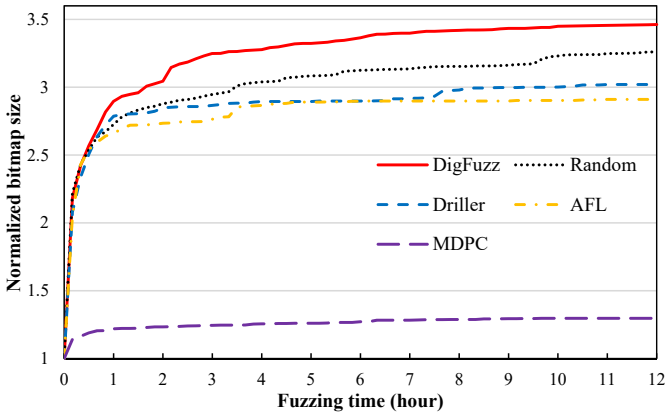


Fig. 6: Normalized bitmap size on CQE dataset

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.

Figure 6 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 12 hours, the *normalized bitmap sizes* in DigFuzz, Random, Driller, and AFL are 3.46 times, 3.25 times, 3.02 times and 2.91 times larger than the base, respectively. Taking the *normalized bitmap sizes* AFL that is aided by dummy concolic execution as a baseline, the concolic execution in in DigFuzz, Random and Driller contributes to discovering 18.9%, 11.7%, and 3.8% more code coverage, respectively.

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 [39]. 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 "demand launch" strategy described in Section II can considerably affect the concolic execution. Third, by comparing DigFuzz 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% vs. 3.8%) and Random (18.9% vs. 11.7%). This fact demonstrates the effectiveness of our strategy in term of code exploration.

We can also see that MDPC is even worse than AFL.

TABLE II: Number of discovered vulnerabilities

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

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 [42], 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 runtime 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 cost of different techniques.

Unlike MDPC that estimates the cost 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.

2) Number of Discovered Vulnerabilities: We then present the numbers of vulnerabilities discovered by all four configurations via Table II. The 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.

This table could further exhibit the effectiveness of DigFuzz by comparing 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 [39]. 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

TABLE III: Performance of concolic execution

	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	51	1,679	5
	49	1709	12	153	2,375	4
	51	877	13	95	1,905	4

for DigFuzz. This means DigFuzz is able to perform similarly with only half of the running time (12 hours vs. 24 hours) and much less hardware resources (2 fuzzing instances per binary vs. 4 fuzzing instances per binary).

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.

The number of inputs generated by the concolic execution and later imported by the fuzzer indicates the contribution of concolic execution to the fuzzer. Therefore, we analyze each imported input, trace back to its source and present numbers in Table III.

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 fuzzing 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 is of much better quality in general.

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 vs. 5) than that in Driller.

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

E. Evaluation on the LAVA dataset

In this section, we demonstrate the effectiveness of our approach on the `LAVA-M` dataset.

1) Discovered vulnerabilities: The LAVA dataset is widely adopted for evaluation in recent studies [12], [27], [32]. A recent report [14] shows that the fuzzing on binaries in `LAVA-M` can be assisted by extracting constants from these binaries and constructing dictionaries for AFL. According to this report [14], we analyze every binary for 5 hours with and without dictionaries respectively.

The discovered vulnerabilities are shown in Table IV. It shows that with dictionaries, the four techniques, DigFuzz, Random, Driller 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 [15], the reason why concolic execution cannot make much contribution for $md5sum$ and $uniq$ are hash functions and unconstrained control dependency. This 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, 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 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 less seeds than DigFuzz and Random. Then the number of paths along traces 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 less bugs than DigFuzz and Random.

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 [14], especially for $base64$, $md5sum$, and $uniq$. Consequently, the code coverage

TABLE IV: Number of discovered vulnerabilities

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

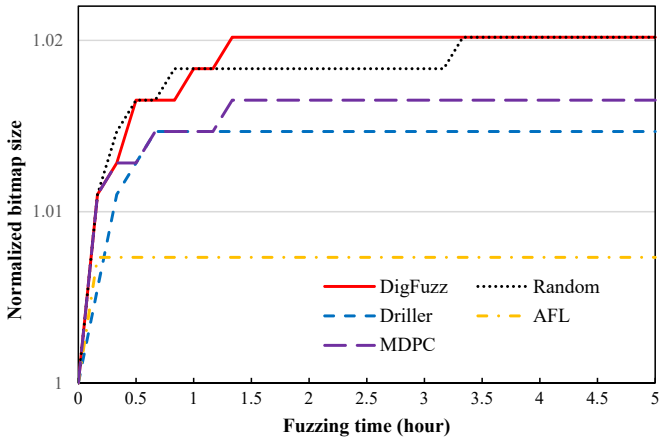


Fig. 7: Normalized incremental bitmap size on LAVA dataset

generated by these fuzzing systems (except MDPC due to the reduced throughput) will be about the same if dictionaries are used. As a result, we present the code coverage without dictionary.

From the Figure 7, we can observe that DigFuzz can cover more code than the other three configurations. Both of the DigFuzz and Random outperform MDPC and Driller, and all the hybrid systems (DigFuzz, Random, MDPC and Driller) is stably better than AFL.

The code coverage in Figure 7 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 scale of the execution trees generated from the programs in LAVA-M are small and contains only a few execution paths. Thus, path prioritization (the core part in DigFuzz) can not contribute much since there exists no path explosion problem.

F. Evaluation on real programs

We also attempt to evaluate DigFuzz on large real-world programs. However, we observe that the symbolic engine, angr, used in DigFuzz, does not have sufficient support for analyzing large real-world programs. This is also supported by a recent fuzzing research [32]. More specifically, the current version of angr lacks support for many system calls and cannot make any progress once it reaches an unsupported system call. We test DigFuzz on more than 20 real-world programs, and the results show that the symbolic engine on average can only execute less than 10% branches in a whole execution trace before it reaches an unsupported system call. We argue that the scalability of symbolic execution is a different research area which is orthogonal to our focus in this paper. We will leave the evaluation of real-world programs as future work.

G. Case study

In this section, we demonstrate the effectiveness of our system by presenting an in-depth comparison between DigFuzz and Driller 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 [39].

Figure 8 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 $check\_3$.

```
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. 8: The vulnerability in $NRFIN\_00017$.

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 three hybrid fuzzing systems 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 fuzzing. Further inspection shows that the concolic executor had to process 7, 23 and 94 inputs retained by the fuzzer for the three condition checks in order to solve the three branches. Eventually, it took 2590 seconds for Driller to generate a

satisfying input for $check\_3$, and guide the fuzzer to reach the vulnerable code. Not surprisingly, DigFuzz and Random were able to trigger the vulnerability much faster, in 691 and 1438 seconds.

Unlike Driller, both Random 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 DigFuzz selects the paths with the highest priority. Figure 10 and Figure 11 show the time stamps when concolic executions were invoked and how they helped fuzzing in DigFuzz and Random. As we can see from the figures, DigFuzz performed 7 concolic executions in 691 seconds while Random finished 26 executions in 1438 seconds before they could generate inputs to satisfy $check\_3$ and help fuzzing reach the vulnerability. In terms of the input generation, DigFuzz managed to generate 96 inputs within which 37 were imported by fuzzing. Random, on the other hand, generated 373 inputs and 44 of them were imported. Moreover, by the time of the 691st second, DigFuzz generated 37 imported inputs while Random can only generate 4. These numbers show that DigFuzz could generate inputs with much higher quality than Random.

2) In-depth analysis: We further present the details on how concolic execution helped the fuzzer to bypass these three checks in DigFuzz. Figure 9 briefly shows the execution tree for $NRFIN\_00017$. In the figure, the path that leads to the vulnerability is shown as the red path. 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 and Driller quickly generated a satisfying input to bypass $check\_1$, because the current execution tree is pretty small as shown in Figure 9. After this, the fuzzers resumed from the stuck state. It went into *do_build()*, quickly generated 23 interesting inputs in less than 1 minute and then reached the $check\_2$. The concolic executor in DigFuzz prioritized the 23 inputs, accurately selected the one that corresponded to $check\_2$ 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 Figure 9. At this moment, the concolic executor was handed with 97 inputs from the fuzzer and had to pick the right one to reach $check\_3$. From Figure 10, we can observe that DigFuzz took only 2 concolic executions to bypass $check\_3$. And eventually, DigFuzz reached the vulnerability at the 691st second.

To further demonstrate the effectiveness of path prioritization, we examined how the concolic executors work in DigFuzz, Random, and Driller. In particular, when the fuzzer had bypassed $check\_2$, it quickly discovered more blocks thus retained amount of inputs. As shown in Figure 9, 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 Figure 9, the fuzzer quickly got blocked at $check\_3$. Driller will not identify this specific branch until the the fuzzer gets stuck again. Random requires to go through all the covered paths for discovering missed 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, 3 times in Random, and only 1 time in DigFuzz. This result indicates that the path prioritization in DigFuzz was 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.

VI. Discussion

A. Threats to Validity

Our experimental results are based on the limited dataset presented in the paper. Efforts have been made to evaluate DigFuzz on real programs, but Angr [38] fails to symbolically execute programs whenever it encounters an unsupported system call. Therefore, the results may not be fully representative of real-world programs. An evaluation on such programs is necessary to draw conclusions on the effectiveness of the proposed techniques in practice. We will leave the evaluation of real-world programs as our future work.

B. Limitations

First, although the "discriminitive 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. However, based on our evaluation, we can see the throughput reduction for fuzzer is negligible. Moreover, since each node in the tree only carries very limited information, the total memory consumption of the execution tree is very manageable.

Second, since DigFuzz only estimates the difficulty of paths for fuzzer to explore but does not consider the complexity of constraint solving, it is possible that the constraints collected from the picked path can be unsolvable which may result in a waste of concolic execution cycle. In addition, it is also possible that the most promising path which could lead to a vulnerability is not the hardest path picked by DigFuzz. These two limitations are due to our model of finding the right path to explore. We consider solving them as future work.

VII. Related Work

Fuzzing and symbolic execution are the two mainstream techniques for program testing. Many prior efforts have been made to improve them [3], [27], [33], [36]. BuzzFuzz [17] leverages dynamic tainting to identify inputs bytes that are processed by suspicious instructions. Dowser [20] employs reverse engineering techniques to identify input fields that are concerned with suspicious functions. Vuzzer [34] leverages control- and data-flow features to accurately determine where and how to mutate such inputs. Skyfire [40] leverages the knowledge in the vast amount of existing samples to generate well-distributed seed inputs for fuzzing programs. Angora [12] aims to increase branch coverage by solving path constraints without symbolic execution. T-Fuzz [32] firstly allows the fuzzer to work on the transformed program by removing

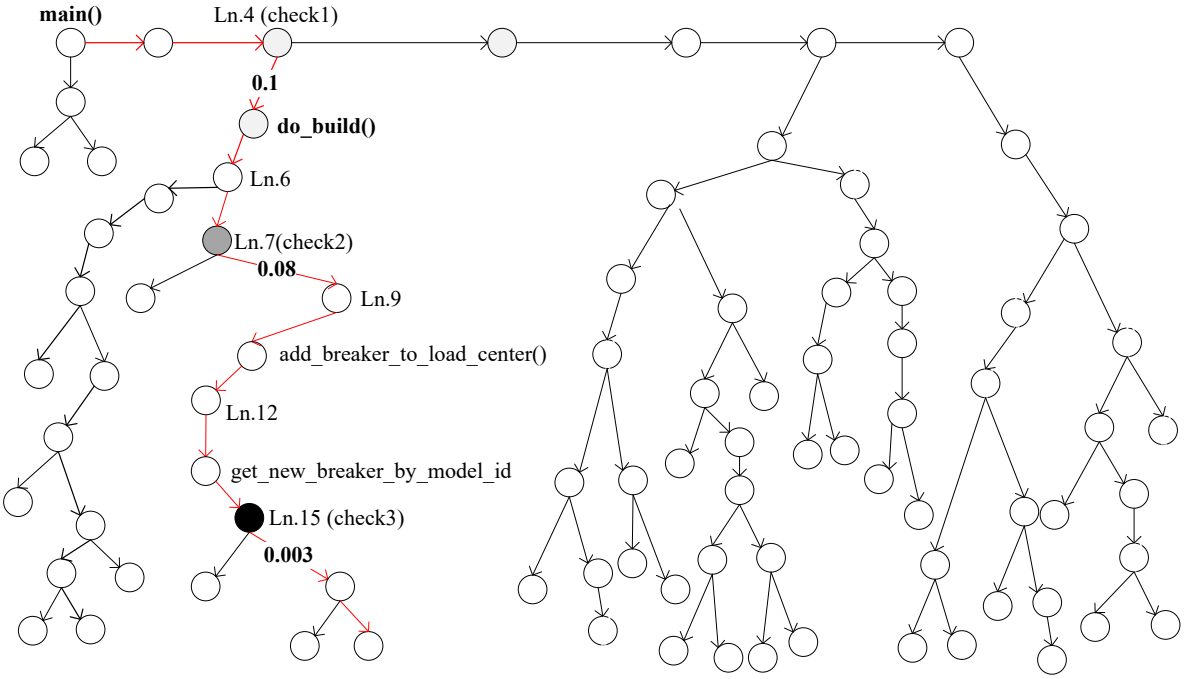


Fig. 9: The execution tree for $NRFIN\_00017$.

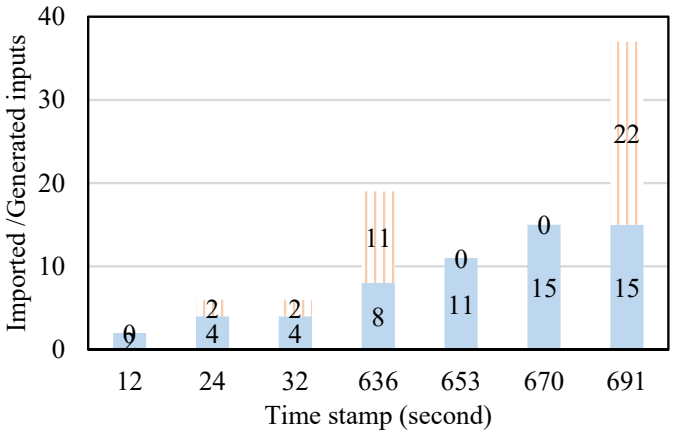


Fig. 10: Concolic executions by DigFuzz on $NRFIN\_00017$.

sanity checks that the fuzzer fails to bypass. As an auxiliary post-processing step, T-Fuzz leverages a symbolic execution-based approach to filter out false positives. CollAFL [16] 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. Veritesting [1] tackles the path explosion problem by employing static symbolic execution to amplify the effect of dynamic symbolic execution. Mayhem [10] proposes to combine on-line and off-line symbolic execution to deal with the problem of exhausted memory. The main contribution of DigFuzz is to propose a more effective strategy to combine fuzzing with concolic execution. Therefore, the advances of fuzzing and concolic execution are out of our scope.

Hybrid fuzzing system. Most of hybrid fuzzing systems follow the observation to augment fuzzing with selective symbolic execution [29], [39], [41]. Both Driller and hybrid concolic testing deploy a "demand launch" strategy. Instead, DigFuzz designs a novel "discriminative dispatch" strategy to better utilize the capability of concolic execution. TaintScope [41] deploys dynamic taint analysis to identify the checksum check points and then applies symbolic execution to generate inputs satisfying checksum. TaintScope is specifically designed for dealing with checksum, and DigFuzz has a more general scope. More important, DigFuzz employs the Monte Carlo model to estimate probabilities and prioritize paths, which is more lightweight than the dynamic taint analysis.

MDPC [42] proposes a theoretical framework for optimal concolic testing. It defines the optimal strategy based on the probability of program paths and the cost of constraint solving, which is similar with our insight to identify path probabilities. In contrast to MDPC [42] that adoptes heavyweight techniques to calculate the cost of fuzzing and concolic execution, our model calculates probabilities with coverage statistics, which is more lightweight and practical.

QSYM [46] integrates the symbolic emulation with the native execution using dynamic binary translation. It also alleviates the strict soundness requirements of conventional concolic executors to achieve better performance as well as to make it scalable to real-world programs. The main focus for QSYM is to improve the efficiency of concolic execution while our approach tries to make better use of concolic executor by selectively dispatching only the hardest work to it.

Another type of hybrid fuzzing system is to regard the symbolic execution as a guide for input generation or path selection. Pak [31] 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

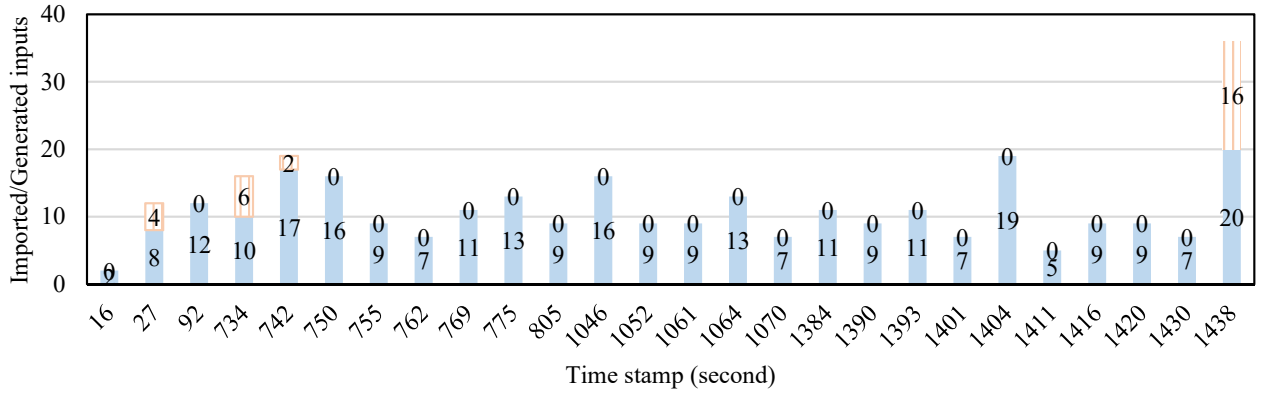


Fig. 11: Concolic executions by Random on $NRFIN\_00017$.

execution to assign probabilities to program paths, and then takes these probabilities to guide the path exploration in fuzzing.

Path prioritization in symbolic execution. Path prioritization is promising for mitigating the path explosion problem in dynamic symbolic execution. Representative studies include heuristics techniques and sound program analysis techniques [9]. These heuristics include using the control-flow graph to guide the exploration, frequency-based and random-based techniques [6]–[8]. Recently, path prioritization is adopted to combine with evolutionary search, in which a fitness function is defined to guide the symbolic execution [2]. Compared with these path exploration techniques, the path prioritization in DigFuzz is to prioritize paths with probabilities how difficult for fuzzing to pass through. To the best of our knowledge, we are the first to investigate the path prioritization problem in the hybrid fuzzing system.

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 [37], [45]. 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], [11], [44] by prioritizing seed inputs. Woo et al. [44] model black-box fuzzing as a multi-armed bandit problem where the energy of a seed is computed based on whether or not it has exposed a crash in any previous fuzzing iteration. AFLfast [4] improves the seed selection strategy of AFL by assign more energy to inputs that are less frequently taken by AFL. 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 technique to offload the fuzzer with paths that are difficult to explore.

Test case prioritization attempts to reorder test cases in a way that increases the rate at which faults are detected [21], [22], [24], [26], [28]. The path prioritization in this study is to obtain the missed paths that are most likely to block the fuzzer. The search algorithm is also closely related to the search-based test prioritization and other search-based software engineering [23].

VIII. Conclusion

In this paper, we perform a thorough investigation on some state-of-the-art hybrid fuzzing systems and point out several fundamental limitations in the "demand launch" and "optimal switch" strategies deployed in these systems. We further propose a "discriminative dispatch" strategy to better utilize the capability of concolic execution by designing a Monte Carlo based probabilistic path prioritization model to quantify each path's difficulty. We implement a prototype system DigFuzz based on the design and conduct comprehensive evaluation using two popular datasets. 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 system Driller and MDPC.

Acknowledgment

We thank our shepherd Endadul Hoque and the anonymous reviewers for their insightful comments on our work. This work is partly supported by National Science Foundation under Grant No. 1664315, Office of Naval Research under Award No. N00014-17-1-2893, and National Natural Science Foundation of China under Grant No.61672394 and 61872273. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 1083–1094.

[2] A. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P. Tonella, and T. Vos, "Symbolic search-based testing," in *the 26th International Conference on Automated Software Engineering*, 2011, pp. 53–62.

[3] M. Bohme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of CCS'17*, 2017, pp. 2329–2344.

[4] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based Greybox Fuzzing as Markov Chain," in *Proceedings of CCS'16*, 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*. Springer, 2016, pp. 25–34.

[6] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *the 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2008, pp. 443–446.

[7] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*. USENIX Association, 2008, pp. 209–224.

[8] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler, "EXE: automatically generating inputs of death," *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 2, pp. 1–38, 2008.

[9] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[10] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing Mayhem on Binary Code," in *2012 IEEE Symposium on Security and Privacy*, 2012, pp. 380–394.

[11] S. K. Cha, M. Woo, and D. Brumley, "Program-Adaptive Mutational Fuzzing," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 725–741.

[12] P. Chen and C. Hao, "Angora: Efficient fuzzing by principled search," in *39th IEEE Symposium on Security and Privacy*. IEEE, 2018.

[13] DARPA, "Cyber Grand Challenge Challenge Repository," 2017, http://www.lungetech.com/cgc-corpus/.

[14] B. Dolan-Gavitt, "Of Bugs and Baselines," 2018, http://moyix.blogspot.com/2018/03/of-bugs-and-baselines.html.

[15] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "LAVA: Large-Scale Automated Vulnerability Addition," in *2016 IEEE Symposium on Security and Privacy*, 2016, pp. 110–121.

[16] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *39th IEEE Symposium on Security and Privacy*. IEEE, 2018.

[17] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 474–484.

[18] J. Geldenhuys, M. B. Dwyer, and W. Visser, "Probabilistic symbolic execution," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 2012, pp. 166–176.

[19] W. R. Gilks, "Markov chain monte carlo," *Encyclopedia of Biostatistics*, 2005.

[20] I. Haller, A. Slowinska, H. Bos, and M. M. Neugschwandtner, "Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violation," in *the 22nd USENIX conference on Security*, 2013, pp. 49–64.

[21] D. Hao, L. Zhang, L. Zhang, G. Rothermel, and H. Mei, "A unified test case prioritization approach," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 24, no. 2, p. 10, 2014.

[22] D. Hao, L. Zhang, L. Zang, Y. Wang, X. Wu, and T. Xie, "To be optimal or not in test-case prioritization," *IEEE Transactions on Software Engineering*, vol. 42, no. 5, pp. 490–505, 2016.

[23] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, p. 11, 2012.

[24] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon, "Comparing white-box and black-box test prioritization," in *the 38th International Conference on Software Engineering*, 2016, pp. 523–534.

[25] X. Jia, C. Zhang, P. Su, Y. Yang, H. Huang, and D. Feng, "Towards efficient heap overflow discovery," in *Proceedings of the 26th USENIX Security Symposium*. USENIX Association, 2017, pp. 989–1006.

[26] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov, "An extensive study of static regression test selection in modern software evolution," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 583–594.

[27] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: program-state based binary fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 627–637.

[28] Y. Lu, Y. Lou, S. Cheng, L. Zhang, D. Hao, Y. Zhou, and L. Zhang, "How does regression test prioritization perform in real-world software evolution?" in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 535–546.

[29] R. Majumdar and K. Sen, "Hybrid Concolic Testing," in *29th International Conference on Software Engineering (ICSE'07)*, 2007, pp. 416–426.

[30] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[31] B. S. Pak, "Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution," *Masters thesis, School of Computer Science Carnegie Mellon University*, 2012.

[32] h. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation," in *39th IEEE Symposium on Security and Privacy*. IEEE, 2018.

[33] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2155–2168.

[34] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing," in *2017 Network and Distributed System Security Symposium*, 2017.

[35] C. P. Robert, *Monte carlo methods*. Wiley Online Library, 2004.

[36] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kafl: Hardware-assisted feedback fuzzing for os kernels," in *USENIX Security 17*, 2017, pp. 167–182.

[37] H. Seo and S. Kim, "How we get there: a context-guided search strategy in concolic testing," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 413–424.

[38] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy*, 2016, pp. 138–157.

[39] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," in *Proceedings 2016 Network and Distributed System Security Symposium*, 2016.

[40] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *38th IEEE Symposium on Security and Privacy (S&P 2017)*, 2017.

[41] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *2010 IEEE Symposium on Security and Privacy*, 2010, pp. 497–512.

[42] X. Wang, J. Sun, Z. Chen, P. Zhang, J. Wang, and Y. Lin, "Towards optimal concolic testing," in *ICSE 2018*.

[43] Wikipedia, "Rule of three (statistics)," 2017, https://en.wikipedia.org/wiki/Rule_of_three_%28statistics

[44] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 511–522.

[45] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *IEEE/IFIP International Conference on Dependable Systems & Networks.*, 2009, pp. 359–368.

[46] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing," in *Proceedings of the 27th USENIX Security Symposium*, 2018.

[47] M. Zalewski, "American Fuzzy Lop," 2017, http://lcamtuf.coredump.cx/afl/.