Fuzzle: Making a Puzzle for Fuzzers

Haeun Lee
KAIST
Daejeon, Korea
haeun.lee@kaist.ac.kr

Soomin Kim
KAIST
Daejeon, Korea
soomink@kaist.ac.kr

Sang Kil Cha
KAIST
Daejeon, Korea
sangkilc@kaist.ac.kr

ABSTRACT

With rapidly growing fuzzing technology, there has been surging demand for automatically synthesizing buggy programs. Previous approaches have been focused on injecting bugs into existing programs, making them suffer from providing the ground truth as the generated programs may contain unexpected bugs. In this paper, we address this challenge by casting the bug synthesis problem as a maze generation problem. Specifically, we synthesize a whole buggy program by encoding a sequence of moves in a maze as a chain of function calls. By design, our approach provides the exact ground truth of the synthesized benchmark. Furthermore, it allows generation of benchmarks with realistic path constraints extracted from existing vulnerabilities. We implement our idea in a tool, named Fuzzle, and evaluate it with five state-of-the-art fuzzers to empirically prove its value.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; *Software verification and validation.*

ACM Reference Format:
Haeun Lee, Soomin Kim, and Sang Kil Cha. 2022. Fuzzle: Making a Puzzle for Fuzzers. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3551349.3556908

1 INTRODUCTION

Despite rapidly growing fuzzing technology [47], there has been limited study on automatically generating effective benchmarks for fuzzers. Indeed, most benchmarks used in past research consist of existing buggy programs. For example, Google's Fuzzer Test Suite [32] and FuzzBench [52] incorporate dozens of open-source projects containing previously-known bugs.

However, such benchmarks suffer from several drawbacks. First, they can hardly evolve without significant manual effort, thereby impeding large-scale evaluation. To add a program to a benchmark, one needs to manually analyze the program to see if there is a non-trivial and triggerable bug. Next, as benchmarks tend to remain intact, one can create a fuzzer that *overfits* an existing benchmark. For example, one could design a fuzzer that is optimized only for FuzzBench, so the fuzzer does not perform well in other benchmarks. Finally, such handmade benchmarks do *not* provide the ground truth. That is, one cannot know *a priori* how many bugs there are in each program nor where those bugs are, making it difficult to evaluate the effectiveness of fuzzers.

These observations have spurred research on synthesizing fuzzer benchmarks. LAVA [19], EvilCoder [56], Apocalypse [58], and FixReverter [63] are representative tools that enable on-demand fuzzing benchmark generation, thereby allowing a large-scale evaluation of fuzzers. We refer to such a tool as a *bug synthesizer*. All the existing bug synthesizers take in a program as input, and modify the program by injecting synthetic bugs.

However, such injection-based methods still lack in providing precise ground truth because there is no guarantee that the original programs are bug-free. Besides, bug injection may accidentally change the semantics of the original programs, which can introduce *unintended* bugs. In this regard, previous bug synthesizers are insufficient for rigorously evaluating fuzzers. As noted by Klees *et al.* [37], bug deduplication (or triaging) employed by modern fuzzers is largely inaccurate. Therefore, having the precise ground truth of the benchmark is imperative in evaluating fuzzers.

To address this challenge, we propose a novel bug synthesis algorithm, which automatically generates a buggy C program from scratch. Since we have full control over how a program is synthesized, we know by design the precise ground truth about the resulting programs. By ground truth, we mean the existence and location of bug(s) in each synthesized program.

The key intuition of our approach is to cast the problem of synthesizing bugs as a maze generation problem. Our algorithm outputs a program that behaves similarly to an agent in a maze puzzle, where the goal is to find the exit of the maze. The agent takes in a sequence of actions as input, and moves its location according to the user input. The program terminates when the agent consumes all the user input before reaching the exit. When the agent arrives at the exit, the program aborts and issues a notification that the bug is found. Therefore, as long as there is a feasible execution path to the exit, we can assure that the program has exactly one bug.

There has been longstanding recognition that software testing is similar to maze exploration in that they have the common goal of finding a specific path [28, 44]. In our case, synthesizing a buggy program is equivalent to creating a maze that has a single exit point. Finding a bug here means discovering a path from the entry to the exit point in the maze. There are indeed many practical algorithms to randomly generate complex mazes [59].

The maze-based bug synthesis provides an additional benefit as it enables an effective visualization of the generated benchmarks. As every cell-to-cell movement in a maze directly corresponds to a function call in a generated program, we can readily visualize the program as well as the current progress of a fuzzer in an intuitive manner. For example, the analyst can see which part of a maze is

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10.
https://doi.org/10.1145/3551349.3556908

covered by a fuzzer, and know in which program point the fuzzer is stuck.

Furthermore, to ensure that finding the bug in our synthesized program well resembles finding real bugs, we leverage the path constraints obtained from existing vulnerabilities. Specifically, we use symbolic execution to extract buggy path constraints from CVEs in the SMT-LIB format, and use these constraints to construct program paths that lead to the bug site. In addition to the realistic path constraints, our approach generates programs that resemble human-written code. Specifically, it generates functions that are chained in a recursive manner, similar to those of recursive descent parsers found in modern compilers such as GCC [31].

In this paper, we implement these ideas in a tool, named Fuzzle, and evaluate it with five state-of-the-art fuzzers. In summary, we make the following contributions in this paper.

- We present a novel bug synthesis algorithm that generates a buggy program from scratch.
- We analyze requirements for generating synthetic bugs and show why our algorithm satisfies them.
- We design and implement a novel bug synthesizer, named Fuzzle, and evaluate it with five state-of-the-art fuzzers.
- We publicize our tool and dataset in support of open science: https://github.com/SoftSec-KAIST/Fuzzle

2 RESEARCH GOALS

Past research has identified several design goals for bug synthesizers [19, 36, 56, 58], but we found those goals are insufficient to produce effective benchmarks because they are mainly focused on the characteristics of the synthesized bugs. For example, previous synthesizers do not provide precise ground truth for their output as they have to modify existing programs. Furthermore, the generated benchmarks do not provide any visual aid, making it difficult for a comprehensive analysis of the progress of a fuzzing campaign.

Thus, we introduce two design goals (G7 and G8) that address the testing aspect of generated benchmarks in addition to six existing goals we identified from previous papers [19, 36, 56, 58].

(G1) Unbiased. Our tool should generate *unbiased* bugs in that they do not favor a specific bug detection tool. The synthesized bugs should be created in a way that is independent of the evaluated techniques for a *fair* evaluation.

(G2) Deep. Our tool should produce *deep* bugs, which can only be reached by penetrating a large number of branches.

(G3) Rare. Our tool should create *rare* bugs, which can be triggered by only a small fraction of possible inputs.

(G4) Uncorrelated. Our tool should generate *uncorrelated* bugs such that the injected bugs behave independently of each other. That is, finding one of the bugs should not change the detection probabilities of other injected bugs in the program.

(G5) Reproducible. Our tool should synthesize *reproducible* bugs. That is, there should exist a concrete input to discover the synthesized bugs.

(G6) Realistic. Our tool should produce a bug that can be triggered by exercising a *realistic* path. Since it is controversial as to which bug is realistic, we say a bug is realistic if it can be triggered by exercising a buggy path that resembles the ones of real-world vulnerabilities, e.g., CVEs.

Table 1: Comparison of existing bug synthesizers.

	Existing Goals						New Goals	
Tool Name	**Unbiased** G1	**Deep** G2	**Rare** G3	**Uncorrelated** G4	**Reproducible** G5	**Realistic** G6	**Providing Ground Truth** G7	**Visualizable** G8
LAVA [19]	✗	✓	✓	✓	✓	✗	✗	✗
EvilCoder [56]	✓	✓	✓	Ltd.†	✗	✓	✗	✗
Apocalypse [58]	✓	✓	✓	✓	✓	✗	✗	✗
Bug-Injector [36]	✓	✓	✓	✓	✓	✓	✗	✗
FixReverter [63]	✓	✓	✓	Ltd.†	Ltd.†	✓	✗	✗
Fuzzle	✓	✓	✓	✓	✓	✓	✓	✓

† Limited support.

(G7) Providing Ground Truth. Our tool should provide ground truth, such as the number of bugs and their locations in the synthesized programs, to enable precise fuzzer evaluation.[1]

(G8) Visualizable Progress. Our tool should provide *visual* feedback to fuzzers so that an analyst can intuitively figure out the current progress of a fuzzing campaign.

Table 1 compares Fuzzle against existing bug synthesizers in terms of the eight goals we seek to achieve. LAVA [19] generates bugs that are neither unbiased (G1) nor realistic (G6) due to the nature of the magic value checks it uses. EvilCoder [56] does not always generate reproducible bugs (G5). Apocalypse [58] does not produce realistic bugs (G6) as it injects bugs that are triggered when arbitrarily generated conditions are met. Bug-Injector [36] accomplishes the existing goals (G1–G6), but not the newly proposed goals (G7 and G8). FixReverter [63], the most recent work, injects realistic bugs but some of the injected bugs are not triggerable (G5) and some are triggerable only in combination with other bugs (G4). On the other hand, Fuzzle achieves all the aforementioned goals.

3 OVERVIEW

Fuzzle is inspired by an observation that finding a software bug is analogous to solving a maze—both involve exercising a specific path. Therefore, the problem of synthesizing a buggy program can be cast as a problem of generating a random maze.

The key intuition of our approach is to encode a path in a maze into a chain of function calls in a program. Thus, every synthesized function corresponds to a cell in the maze, and the program runs by hopping around the maze. The program raises a signal, indicating a bug, when the execution reaches a specific target cell, i.e., a function. Thus, finding a buggy path to reach the *buggy function* is equivalent to discovering a path in the maze from its entry to the target cell. For simplicity, we assume that every maze we generate has a single entrance and exit, and each exit cell is our target, although Fuzzle provides an option to select an arbitrary target cell.

[1]The term "ground truth" used by Kashyap *et al.* [36] should not be confused with our definition of (G7). Their ground truth property is equivalent to our reproducible property (G5).

Algorithm 1: Fuzzle overview.

```
1 function Fuzzle(C_algo, C_size, C_seed, C_cycle, C_smt)
2     maze ← MazeGen(C_algo, C_size, C_seed)
3     g ← GraphGen(C_cycle, maze)
4     t ← TemplateGen(g)
5     prog ← ProgRender(C_smt, t)
6     return prog
```

At a high level, Fuzzle takes in five user-configurable parameters (C_{algo}, C_{size}, C_{seed}, C_{cycle}, and C_{smt}) as input, and returns a synthesized C program as output. Each configuration parameter helps customize the output program, e.g., varying the shape of the maze, providing guidance in generating path conditions, and so forth as we will discuss in this section.

Algorithm 1 describes the overall workflow of Fuzzle, which operates in four major steps: (1) maze generation, (2) graph generation, (3) template generation, and (4) program rendering. Fuzzle first generates a random maze (§3.1), transforms the maze into a graph (§3.2), creates a code template (§3.3), and fills in the holes in the template to produce a C program (§3.4).

3.1 Maze Generation (MazeGen)

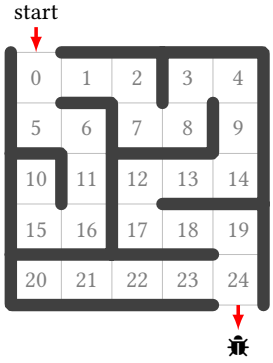
In this step, MazeGen automatically generates a random 2D maze using a maze generation algorithm C_{algo} specified by the user. The algorithm outputs a random maze with a single entry and exit by taking the following configuration parameters as input: (1) C_{size} decides the size of the maze, and (2) C_{seed} specifies the seed used by the pseudorandom number generator in C_{algo}.

For example, Figure 1a illustrates a randomly generated 5x5 maze, where each cell is assigned with a unique identifier starting from 0. In our current implementation, each cell has four edges, meaning that there is a maximum of four ways to enter or exit a cell. This makes visualizing our maze straightforward. By default, MazeGen sets the exit node, i.e., the cell 24, as our target, although any other cells can be a target, too. It also places the entry and exit node at two opposite corners of the maze. Note there are arbitrarily many walks from the entry to the exit. Hence, there are also many execution paths to reach the bug in the synthesized program.

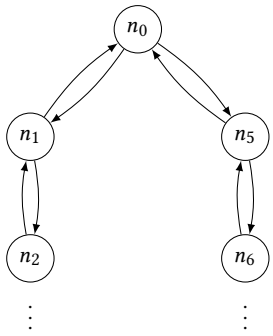
Fuzzle represents a maze using a two-dimensional array (denoted as `maze`) containing information about every edge (i.e., a passage between two cells) in the maze. Specifically, each row of `maze` corresponds to a cell ID, and each column represents one of the four cardinal points (NSEW). Let us use `open` and `close` to mean an open passage and a closed wall, respectively. We can then represent the edge from the cell 0 to cell 5 as `maze[0][`S`] = open`. For cell 12 in Figure 1a, `maze[12][`N`] = close`, `maze[12][`S`] = open`, `maze[12][`E`] = open`, and `maze[12][`W`] = close`. This array, i.e., `maze`, is used in the next step to construct a graph. We detail the implementation of MazeGen in §4.1.

3.2 Graph Generation (GraphGen)

Given the maze array (`maze`) obtained from the previous step, GraphGen now converts the array into a directed graph g where each node corresponds to a cell in the maze, and each edge corresponds to an open passage between two cells in the maze. This step operates with the user configuration C_{cycle}, which controls the



(a) Random 5x5 maze.



(b) Generated graph.

```
1  void func_bug(char *input, ...) { abort(); }
2  void func_0(char *input, ...){
3    ____________ // computation for setting a variable used in conditionals
4    if (__________) func_1(input, ...);
5    else if (__________) func_5(input, ...);
6    else fatal_error("This should never happen");
7  }
8
9  /* func_1, func_2, ..., func_9 are omitted */
10
11 void func_10(char *input, ...){
12   ____________ // computation for setting a variable used in conditionals
13   if (__________) func_15(input, ...);
14   else fatal_error("This should never happen");
15 }
16
17 /* func_11, func_12, ..., func_23 are omitted */
18
19 void func_24(char *input, ...){
20   ____________ // computation for setting a variable used in conditionals
21   if (__________) func_bug(input, ...);
22   else if (__________) func_19(input, ...);
23   else if (__________) func_23(input, ...);
24   else fatal_error("This should never happen");
25 }
26
27 /* main is omitted */
```

(c) Generated program template.

Figure 1: Our example maze.

number of cycles in the resulting graph. The presence of cycles, i.e., loops in the synthesized program, will dramatically increase the number of paths to explore, thereby making some fuzzers struggle in finding buggy paths. We further discuss this issue in §4.2.

GraphGen first transforms every row in `maze` into a node in g. It then creates an edge in g for every open passage in `maze`. For example, let n_0 and n_1 be nodes in g representing the cell 0 and cell 1 of the maze, respectively. Since we know there is an open passage between the two cells, i.e., `maze[0][`E`] = open`, we create an edge from n_0 to n_1 in g. Similarly, we create an edge from n_1 to n_0 because `maze[1][`W`] = open`. Finally, GraphGen removes cycles based on C_{cycle}, which specifies the proportion of remaining cycles. In the case of $C_{\text{cycle}} = 100\%$, we do not remove any cycle. When $C_{\text{cycle}} = 0\%$, we remove all the back edges from the graph. Figure 1b presents the resulting graph (in part) when $C_{\text{cycle}} = 100\%$.

3.3 Template Generation (TemplateGen)

Next, TemplateGen takes in the directed graph g as input, and returns a template t as output, which is a C-like code representation

```
1  void func_24(char *input, int idx){
2    int nBytes = 3;
3    ...  // check that input size is large enough
4    int flag = 0;
5    if(!((uint8_t) 95 == input[idx])){
6      if ((uint8_t) 0 <= input[idx + 1]){
7        ...
8        /* complex logic is placed here */
9        ...
10       flag = 1;
11     }
12   }
13   if (flag == 1) func_bug(input, idx + nBytes);
14   else if (input[idx + 2] < 0) func_19(input, idx + nBytes);
15   else if (input[idx + 2] >= 0) func_23(input, idx + nBytes);
16   else fatal_error("This should never happen");
17 }
```

C_{smt}

Figure 2: A function in C program synthesized from the 5x5 maze shown in Figure 1 using CVE-2016-6131 [54].

of the maze. Essentially, t is an incomplete C program with holes, i.e., placeholders, which will be later completed by ProgRender. Figure 1c shows t generated from our example maze where every placeholder is highlighted with a green box.

The generated template t encodes every path in g as a chain of function calls. Specifically, TemplateGen converts every node in g to a function, and every edge to a call between two functions. Each function in t has a name starting with the prefix "`func_`", followed by a cell ID. For example, `func_0` corresponds to the cell 0 (n_0 in g). There are two function calls in `func_0` to `func_1` and `func_5` because n_0 has two outgoing edges to n_1 and n_5. There is only a single function call in `func_10` as it represents a dead-end where the only way out is to go to n_{15}, i.e., `func_15`. Note that `func_15` also contains a function call to `func_10`. There is a `main` function that simply calls `func_0`, but we do not show it for brevity.

Every function call is guarded with a conditional expression (e.g., Line 4, 5, 13, and 21–23 of Figure 1c), which will be populated by ProgRender in the next step. The template also preserves space in Line 3, 12, and 20 for putting more complex logic into the conditionals. As we will discuss in §3.4, ProgRender makes sure that every conditional expression is satisfiable so that the `else` branches in Line 6, 14, and 24 will never be executed. This way, we make every program execution flows only through the open passages, but not through the walls.

Our design allows the functions to be called recursively to form loops. Such a recursive structure is commonly found in recursive descent parsers, such as GCC [29–31]. Benchmarks generated by Fuzzle are indeed quite similar to the C/C++ parsers of GCC. Depending on the value of C_{cycle}, however, the resulting program may not have a loop. For example, when $C_{\text{cycle}} = 0\%$, Line 13 and 14 of Figure 1c will be removed, in which case `func_10` simply terminates program execution because n_{10} is a dead-end.

3.4 Program Rendering (ProgRender)

The final step is to set up a proper path constraint for every passage in the maze by substituting all the placeholders in t with carefully crafted C expressions. Figure 2 shows a part of the final C program synthesized with C_{smt} obtained from CVE-2016-6131 [54]. At a high level, the program moves around the maze by making function calls until it consumes all the user input (like a recursive descent parser)—there is an input size check in every function, e.g., Line 3 in Figure 2. Each function consumes a different number of input bytes based on the inserted conditional expressions. When the program reaches the buggy node, i.e., `func_bug`, the program raises a `SIGABRT` to let a fuzzer recognize the bug.

ProgRender fills in the placeholders first along a buggy path, and then along the rest of the paths. First, it disregards any cycles in g, and finds the shortest path from the entry node to the target node, which corresponds to a buggy path in the final program. It then fills in the placeholders along the path with realistic path constraints C_{smt} obtained from a real program execution. For example, calling `func_bug` from `func_24` is part of the buggy path, whereas calling one of the other two functions, `func_19` and `func_23`, is not. Therefore, Line 13 contains a guard for the buggy path, and the `flag` variable in the guard becomes 1 if and only if all the conditions (Line 5–9) extracted from C_{smt} are satisfied. The key challenge here is to form a condition by extracting expressions from C_{smt} that use the same input byte(s). We discuss further in §4.3.1 how Fuzzle constructs a guard for each function call by subdividing expressions in C_{smt}.

Next, it handles the rest of the placeholders (Line 14–15) by synthesizing path constraints while ensuring the generated program achieves all the aforementioned goals in §2. By default, ProgRender employs equally-divided input range checks to ensure that the generated conditions are always satisfiable—thus, Line 16 will never be executed. The use of range checks ensures that the remaining input space is equally partitioned such that every possible function call that does not lead towards the buggy site is equally probable from a given function. Although such input range checks are simple, they allow Fuzzle to generate benchmarks that are well-suited for evaluating the path exploration ability of fuzzers. ProgRender also supports several other rendering strategies as discussed in §4.3.2.

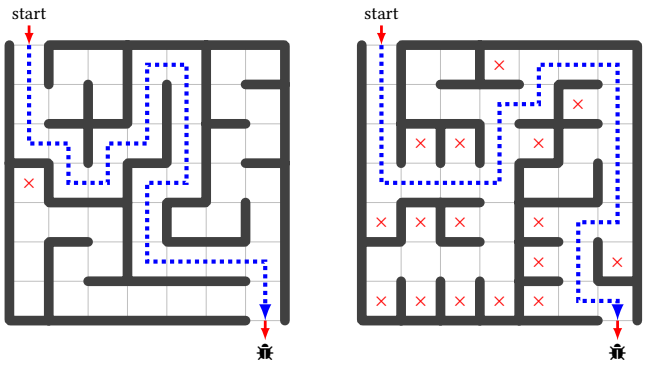
4 DESIGN

This section presents several design choices we made to implement Fuzzle. We also review how Fuzzle achieves all the design requirements, and show its implementation details.

4.1 Maze Generation Algorithms

As the shape of a generated maze determines the flow of the program paths, the choice of a maze generation algorithm can significantly affect the performance of fuzzers. Fuzzle allows the analyst to choose a maze generation algorithm via C_{algo}. Currently, it implements five different maze generation algorithms including backtracking, Kruskal's, Prim's, Sidewinder, and Wilson's algorithm. All these algorithms ensure that there is no unreachable area in the maze [59]. Four of them (backtracking, Kruskal's, Prim's, and Wilson's algorithms) are graph-theoretical methods where maze generation is considered as generating a random spanning tree [18]. In contrast, Sidewinder does not require the entire maze to be stored in memory at once, as it generates one row at a time. Starting with an open passage at the top, it makes subsequent rows by removing randomly chosen walls.

We can easily visualize the differences between the algorithms by comparing the resulting mazes. For example, Figure 3 depicts



(a) Backtracking algorithm (b) Prim's algorithm

Figure 3: Comparison between two mazes generated by two different algorithms.

two different mazes generated by (a) the backtracking algorithm and (b) Prim's algorithm. The maze generated with Prim's algorithm has many short passages that lead to dead-ends, which are labelled with × marks, whereas the maze from the backtracking algorithm has long-winding passages with only a few branching points. Our empirical study shows that such differences indeed affect the performance of tools on the generated benchmarks (§5.4.1).

4.2 Restricting Loop Conditions

Any walks in g form a loop in the generated program when their starting node and ending node are the same. For example, a walk $n_0 \rightarrow n_1 \rightarrow n_0$ in Figure 1b constitutes a loop. Such a loop can make Fuzzle-generated programs run indefinitely. Therefore, it is necessary to control the termination conditions of those loops. One potential solution is to employ a global induction variable to impose a terminating condition for each loop. Currently, we take the following two approaches to handle this issue. We leave it as future work to fine-control loop conditions, though.

First, we allow the user to control the number of cycles in g with C_{cycle}. Decreasing the value of C_{cycle} reduces the number of walks that form a loop in the generated program. A large value of C_{cycle}, on the other hand, leads to an increased number of paths to explore, thereby generating a benchmark program with a hard-to-find bug. We empirically evaluate the impact of C_{cycle} in §5.4.3.

Furthermore, we limit the amount of input the program can take in to prevent an infinite loop. We note that modern fuzzers, including AFL, are unlikely to be stuck in the loops of the program as they often prefer taking infrequently visited nodes. Nevertheless, by making the program consume only a limited amount of input, we ensure that the program always terminates. Specifically, Fuzzle computes the maximum input size of a program to be the total sum of the number of bytes required by each function.

4.3 Rendering Path Conditions

Recall from §3.4, Fuzzle produces a final program by replacing the placeholders in the template with generated conditional expressions. Fuzzle first renders conditions along the shortest solution path from an entry to the buggy function, i.e., buggy path conditions, (§4.3.1). It then renders the conditions for the rest of the paths, i.e., non-buggy path conditions, (§4.3.2).

4.3.1 Buggy Path Conditions. Fuzzle renders the buggy path conditions using realistic path constraints C_{smt} obtained from a real program execution. In our current implementation, we obtain C_{smt} by running KLEE [10] with a concrete input that triggers a previous CVE. Fuzzle expects that C_{smt} is in the standard SMT-LIB format.

We first convert the given path formula C_{smt} into a set of C expressions. We then use those expressions to construct a guard for each function call in the buggy path. We use nested if-statements if the guard has multiple conditions. As the functions consume the input sequentially, the key challenge here is to ensure the expressions that share any variables are used in the same function.

To achieve this goal, we subdivide C_{smt} into independent subformulas that do not share any variables, as in the independent constraint optimization used in symbolic execution [10, 11]. We then insert each subformula into one of the functions along the buggy path. When there are a fewer number of independent subformulas than the total number of functions in the buggy path, we use our basic range checks for the remaining functions (§4.3.2). This allows the generated program to better reflect the difficulty of finding the original bug used in obtaining C_{smt}.

4.3.2 Non-Buggy Path Conditions. Fuzzle employs three different ways to fill in the rest of the placeholders: (1) range-based, (2) equality-based, and (3) real-constraints-based approaches.

First, by default, Fuzzle uses equally-divided range checks. For example, `func_24` in Figure 2 has two function calls (in Line 14 and 15) that are not relevant to the buggy path. They both consume the same input byte at the same offset (`idx + 2`), and equally divide the input space into two. Although simple, this approach is effective in evaluating state-of-the-art fuzzers as our empirical study shows in §5.4. This is because the maze structure with its recursive call chains indeed constitutes complex paths to navigate.

Second, Fuzzle can use equality checks to construct path conditions. Since penetrating an equality condition has been one of the hurdles in grey-box fuzzing until recently [14, 38], we can fill in each placeholder by creating an equality check with a random integer. Our study shows that this strategy effectively varies the bug finding difficulty for several fuzzers (§5.5.1).

Finally, Fuzzle can leverage the realistic constraints C_{smt} to fill in the rest of the conditionals, too. It is indeed trivial to negate a random part of C_{smt} to construct additional conditional expressions. For example, we can make a new conditional expression by negating the condition in Line 5 of Figure 2 while using the same nested conditionals as long as the resulting path constraints are satisfiable. This strategy hinders fuzzers from distinguishing buggy paths from non-buggy paths, consequently preventing them from overfitting to our benchmark. We further evaluate the impact of using this strategy in §5.5.2.

4.4 Visualization

One of the distinguishing features of Fuzzle is that the coverage achievement of any synthesized program is easily visualized in an intuitive maze figure. Since Fuzzle represents a maze using a 2D

array (§3.1), we can easily transform the array into a black-and-white image, where black pixels indicate walls. Using this image, Fuzzle visualizes code coverage progress during a fuzzing campaign. Whenever Fuzzle obtains code coverage reports from a fuzzer, it fills in each cell of the maze figure with appropriate colors. In our current implementation, we use green to indicate the cells that have been visited at least once, and red for the cells that have never been visited. This way, the analysts can intuitively understand the current progress of fuzzers (see §5.8).

4.5 Review of the Design Requirements

Does Fuzzle achieve all the design requirements we defined in §2? We answer this question by recalling the design of Fuzzle.

Fuzzle generates an unbiased fuzzing benchmark (**G1**) in that synthesized bugs do not favor a specific fuzzer. Our empirical study in §5.4.6 confirms this requirement by showing that although there are better performing fuzzers overall, none of the fuzzers shows the best performance in all programs.

Fuzzle also produces deep (**G2**) and rare (**G3**) bugs because they can only be triggered by exercising specific paths that are guarded by a large number of conditionals obtained from real path constraints C_{smt}. Furthermore, one can easily adjust the deepness and the rarity of bugs by varying C_{smt}, C_{size}, and C_{cycle}.

Fuzzle synthesizes uncorrelated bugs (**G4**) because there can only be a single bug in each program by its design. Fuzzle generates reproducible bugs (**G5**) as it always checks the satisfiability of synthesized path conditions before rendering a program. Fuzzle constructs a buggy path by embedding realistic path constraints C_{smt} obtained from a real program execution (**G6**).

Fuzzle provides the precise ground truth of the generated benchmarks (**G7**) as it knows exactly where in the program the injected bug is located and how one should exercise the buggy execution path to trigger it.

Lastly, Fuzzle produces intuitive visual feedback to the user (**G8**) by showing the current progress of a fuzzing campaign or in a post-analysis to learn why each fuzzer struggles to traverse buggy paths (see §4.4). Although a similar feature can be implemented by visualizing Control Flow Graphs, they are often too large to provide concise and intuitive visual feedback to the users. Fuzzle is indeed the first bug synthesizer providing intuitive visual feedback.

4.6 Implementation

We implemented Fuzzle with 1.5K SLoC of Python and 700 SLoC of Bash. We make our implementation of Fuzzle and the generated benchmark publicly available on GitHub. Fuzzle produces random mazes using the `mazelib` Python library v0.9.12 [60], which implements the well-known maze generation algorithms we described in §4.1. To parse SMT formulas (§4.3.1), Fuzzle uses `pySMT` v0.9.0 [22], a Python library for manipulating and solving SMT formulas. For visualization explained in §4.4, Fuzzle uses the `matplotlib` v3.4.2 [27] to create visual representations of the generated programs and code coverage information.

5 EVALUATION

The goal of this section is to answer the following research questions about Fuzzle and benchmarks generated by Fuzzle.

RQ1. Is Fuzzle efficient in terms of synthesizing buggy programs?
RQ2. How do configuration parameters of Fuzzle affect the quality of synthesized bugs and the performance of fuzzers?
RQ3. How do rendering strategies affect the quality of synthesized bugs and the performance of fuzzers?
RQ4. Does Fuzzle generate benchmarks of customizable difficulty?
RQ5. Are fuzzers' performances on Fuzzle-generated benchmarks representative of performances on real-world programs?
RQ6. Does visualization help understand the performance of fuzzers?

5.1 Experimental Setup

5.1.1 Tools Used. We selected five bug finding tools including AFL (v2.57b) [62], AFLGo (`6e87b69a`) [3], AFL++ (v3.13c) [21], Eclipser (`dc7deb96`) [14], and Fuzzolic (`6aa53031`) [5, 6]. These tools cover a wide variety of bug finding techniques including grey-box fuzzing, directed fuzzing, symbolic execution, and hybrid fuzzing. Because each tool supports different fuzzing targets, we divided the tools into two groups: (1) those that operate only on source code, and (2) those that operate on binaries. For the first group, which consists of AFLGo only, we use the C source code of the benchmark. For the second group, we use binaries obtained by compiling the benchmark programs with GCC v9.3.0.

5.1.2 Environments. All the experiments were conducted on 2 servers of the same specification: 88 Intel Xeon E5-2699 v4 CPU cores with 2.20 GHz and 128 GB of memory. We ran each experiment in an isolated Docker container, and assigned one CPU core and 8GB of memory for each experiment run. Every fuzzer was run for 24 hours for each fuzzing campaign, and every experiment was repeated five times.

5.1.3 Tool Configuration. As AFL, AFLGo, AFL++, Eclipser, and Fuzzolic require an initial seed to operate, we used a dummy initial seed that contains a single character 'A'. Note, for AFLGo, the value used for time-to-exploitation was 18 hours, which corresponds to 3/4 of the total run time as recommended by the authors.

5.2 Benchmark

With Fuzzle, we created a fuzzing benchmark that consists of 90 distinct programs synthesized from scratch. Recall that Fuzzle takes in five user parameters as input. Since most parameters are continuous, we consider only a subset of possible values for each parameter as summarized below.

- C_{algo}: five different algorithms described in §4.1.
- C_{size}: 20x20, 30x30, 40x40, and 50x50.
- C_{seed}: 1, 3, 5, 7, 9.
- C_{cycle}: 0%, 25%, 50%, 75%, and 100%.
- C_{smt}: 6 path formulas obtained from 6 different CVEs or $\varnothing$.

For C_{smt}, we used 6 CVEs from GNU Binutils v2.26, which have been widely used for evaluating fuzzers [3, 4, 13]. For each CVE, we ran KLEE [10] with an input known to trigger the CVE to obtain the path formulas in the SMT-LIB format. We denote by $\varnothing$ an empty formula, meaning that one can opt-out C_{smt}.

Even with the subset of the parameter values, we still have too many parameter combinations to consider ($3{,}500 = 5 \times 4 \times 5 \times 5 \times 7$). Therefore, we further reduced the number as follows. We

first set up a default configuration with the following parameter values: C_{algo} = Wilson's, C_{size} = 30x30, C_{cycle} = 25%, and C_{smt} = ∅. Using this default configuration, we then exhaustively changed each parameter value one at a time while fixing the other parameters as the same as the default configuration. This gives us 18 (= 5+4+5+7−3) unique combinations of parameter values (-3 for deduplication). For each parameter combination, we synthesized five programs at random by using five different values for C_{seed}, resulting in a total of 90 distinct programs.

Our benchmark has a total of 90,540 functions, which constitutes 1,304,952 SLoC of C. On average, Fuzzle generated 1,006 functions and 14,499 SLoC per program. Our benchmark has 6× larger SLoC than the LAVA-M benchmark [19], which is built upon GNU Coreutils. We note that Fuzzle can always synthesize larger programs by increasing C_{size} without having to rely on existing programs.

5.3 RQ1: Efficiency of Fuzzle

To evaluate the practicality of Fuzzle, we measured the efficiency of Fuzzle in terms of the time required to generate the benchmark introduced in §5.2. In total, Fuzzle took 923.31 seconds, or 15.39 minutes, to generate all 90 programs in the benchmark on the same server machine we described in §5.1.2 using a single core. This means that, on average, each program was synthesized in about 10 seconds. Moreover, as Fuzzle is not dependent on existing programs, one can easily expand our benchmark to create a larger one. Thus, we conclude that Fuzzle is highly efficient in synthesizing benchmark programs.

5.4 RQ2: Impact of Configuration Parameters

How does each parameter affect the performance of each fuzzer? As our benchmark contains programs synthesized with exhaustively varying only one parameter at a time (§5.2), we can answer this RQ by running the tools on each of the synthesized programs. We ran each tool for 24 hours with five trials on each program, which sums up to 54,000 CPU hours. Table 2 summarizes (1) the branch coverage (measured with Gcov), (2) the percentage of runs that successfully found the bug, and (3) the time taken to find the bug. The reported numbers are the arithmetic mean of 25 experimental runs (5 times per PRNG seed C_{seed}).

5.4.1 Impact of C_{algo}. The C_{algo} column of Table 2 shows the impact of varying C_{algo}, which corresponds to five different maze generation algorithms described in §4.1. First, most tools were able to find the bugs more consistently and quicker on the programs generated with the backtracking algorithm than those generated with Kruskal's and Prim's algorithms. This is because Kruskal's and Prim's algorithms tend to produce more branches to explore compared to the backtracking algorithm. Next, most tools achieved the highest code coverage in the programs generated with Sidewinder algorithm, which most likely owes to the fact that mazes generated with Sidewinder have the entire top row as a long open passage. Therefore, these varying results across algorithms demonstrate that changing C_{algo} has a significant impact on the performance of the tools in terms of both code coverage and bug finding capability.

5.4.2 Impact of C_{size}. The C_{size} column of Table 2 presents the impact of C_{size}, which controls the size of mazes we generate. Overall,

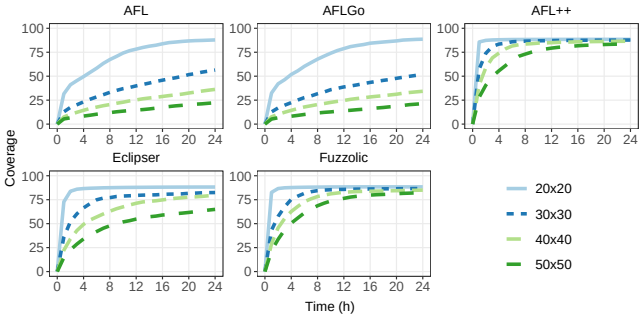


Figure 4: Coverage over time with varying C_{size} for each tool.

all the tools covered less code and took longer to find the bugs as the program size (C_{size}) became larger. We also measured the coverage achievement over time in Figure 4, and observed the same trend. With Mann-Whitney U-Test [1], we verified the significance of C_{size}: p-value was less than 0.001 for all cases.

We note that the performance drop observed for AFL and AFLGo is much larger than that of the other fuzzers. With 20x20 mazes, AFL and AFLGo found bugs in all but one run. As the size increased to 40x40, however, both AFL and AFLGo did not find any bug. This is not surprising because both tools have less effective seed scheduling and mutation strategies compared to the others. These results align well with other comparative studies performed on other real-world benchmarks [21, 46, 52], which indeed signifies the value of Fuzzle as a benchmark generation tool. We further discuss the real-world representativeness of Fuzzle in §5.7.

5.4.3 Impact of C_{cycle}. The C_{cycle} column of Table 2 presents the impact of C_{cycle}, which controls the proportion of cycles to be removed when synthesizing benchmarks. Overall, most tools achieved less coverage and took more time to find the bugs as the number of cycles increased. However, its impact was less than that of C_{size} because the coverage-guidance employed by most of the tools helps them explore less-exercised paths first.

AFL++, Eclipser, and Fuzzolic performed well across all C_{cycle} values, covering more than 75% of the branches and finding the bugs in every experimental run. This is intuitive as their coverage-driven feedback mechanism should guide them to less-explored branches (and thus, call edges). On the other hand, AFL and AFLGo became significantly less effective as the number of cycles increased. This is for the same reasons we discussed in §5.4.2.

5.4.4 Impact of C_{smt}. The C_{smt} column of Table 2 shows the impact of using different path formulas obtained from different CVEs. Unlike other parameters, we observed significant performance differences with varying C_{smt} for every fuzzer. All fuzzers found the bugs, to some degree, in the programs generated with C_{smt} obtained from CVE-2016-4487, CVE-2016-4489, and CVE-2016-4492, whereas only Fuzzolic and AFL++ were able to find the bugs in the programs generated using CVE-2016-4491. Such disparity in performance arises due to the differences in the number and the types of constraints extracted from each CVE. CVE-2016-4491, for example, is a stack overflow bug due to infinite recursion. Therefore, the path formula we obtained from the CVE, i.e., C_{smt}, contains 3× more

Table 2: Performance of different tools on programs of varying parameters.

		C_{algo}					C_{size}				C_{cycle}					C_{smt}						
Measure	Tool	Backtracking	Kruskal's	Prim's	Sidewinder	Wilson's*	20	30*	40	50	0%	25%*	50%	75%	100%	CVE-2016-4487	CVE-2016-4489	CVE-2016-4491	CVE-2016-4492	CVE-2016-4493	CVE-2016-6131	∅*
Coverage (%)	AFL	41.0	64.3	65.6	**76.4**	57.7	**87.8**	57.7	36.2	22.3	**68.2**	57.7	49.5	43.5	40.7	37.8	22.7	25.3	43.0	24.6	21.6	**57.7**
	AFLGo	42.6	61.9	55.4	**70.1**	52.2	**88.6**	52.2	34.3	21.4	**59.0**	52.2	46.6	41.5	41.9	39.9	28.2	32.8	46.1	36.2	24.1	**52.2**
	AFL++	**89.0**	87.8	87.5	88.1	87.7	**88.5**	87.7	86.5	83.8	**88.5**	87.7	87.1	86.9	86.6	75.8	52.0	29.0	85.7	76.1	57.7	**87.7**
	Eclipser	**88.2**	86.7	86.5	87.5	82.2	**88.2**	82.2	79.7	64.9	80.3	82.2	85.7	**86.5**	86.4	74.2	40.5	25.4	78.4	61.1	48.2	**82.2**
	Fuzzolic	75.9	87.7	87.4	**88.5**	86.6	**88.4**	86.6	85.0	82.3	86.7	86.6	**87.1**	84.3	75.6	79.8	65.1	55.4	81.1	77.0	54.7	**86.6**
Bugs (%)	AFL	**100**	4	0	12	64	**96**	64	0	0	**92**	64	24	8	8	12	4	0	12	0	0	**64**
	AFLGo	**100**	0	0	16	32	**96**	32	0	0	**76**	32	28	20	16	8	8	0	16	0	0	**32**
	AFL++	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	84	48	4	**100**	80	68	**100**
	Eclipser	**100**	**100**	**100**	**100**	**100**	**100**	**100**	88	68	**100**	**100**	**100**	**100**	**100**	92	44	0	**100**	80	52	**100**
	Fuzzolic	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	**100**	80	48	**100**	**100**	76	**100**
TTE (h)	AFL	**5.17**	23.05	-	21.52	18.20	**10.51**	18.20	-	-	15.28	18.20	21.07	20.23	**11.32**	21.47	20.71	-	21.42	-	-	**18.20**
	AFLGo	**3.99**	-	-	17.35	20.21	**9.25**	20.21	-	-	**17.05**	20.21	20.64	18.69	18.79	**19.72**	20.35	-	21.72	-	-	20.21
	AFL++	**0.15**	0.77	0.99	0.97	0.57	**0.33**	0.57	2.47	4.39	0.57	0.51	0.56	**0.48**	0.57	2.43	7.86	18.73	1.19	6.26	11.24	**0.51**
	Eclipser	**0.15**	2.48	3.56	3.10	1.50	**0.82**	1.50	8.47	12.31	2.58	1.50	1.43	1.41	**1.25**	6.74	11.78	-	5.46	10.01	12.45	**1.50**
	Fuzzolic	**0.24**	1.34	1.84	1.58	1.10	**0.51**	1.10	4.36	5.28	**0.97**	1.10	1.07	1.12	1.76	1.59	2.59	7.73	1.41	2.41	5.32	**1.10**

The best results in each column are marked with grey background. The best results in each row are marked with bold. The parameter values of the default configuration we used (§5.2) are marked with ∗.

clauses compared to the others, making it difficult for fuzzers to penetrate all the constraints.

5.4.5 Impact of C_{seed}. Recall that our benchmark includes five randomly generated programs for each distinct parameter combination. To understand the impact of C_{seed}, we measured the standard deviations of branch coverage achievements and success rates of finding bugs for each of the 90 programs. As a result, we found that varying C_{seed} does not affect the success rate of finding bugs: the standard deviations were less than 0.01 for all the fuzzers. On the other hand, C_{seed} had a relatively greater impact on the coverage: the standard deviations were around 9.6%. This is because, even with the same parameters, one can obtain significantly different shapes of mazes depending on the PRNG seed.

5.4.6 Is our benchmark unbiased? We note that the results from Table 2 show that the bugs synthesized by Fuzzle are unbiased (§2). We found that there is no single fuzzer that shows the highest performance across all the programs. For example, AFL++ and Eclipser were the fastest in finding bugs from the programs generated with the backtracking algorithm, whereas Fuzzolic was the fastest with different C_{smt}. Thus, we conclude that a benchmark generated by Fuzzle is unbiased as long as it uses different combinations of the configuration parameters to generate the benchmark.

5.5 RQ3: Impact of Rendering Strategies

Recall from §4.3.2 that Fuzzle employs three different strategies to render the non-buggy path conditions. To understand the impact of using those strategies, we generated more benchmark programs with the two non-default strategies: (1) equality-based strategy (§5.5.1), and (2) real-constraints-based strategy (§5.5.2). We then ran all the fuzzers on them, and measured their performance.

5.5.1 Impact of Equality Checks. We first generated programs by replacing increasing proportion of range checks with equality checks (from 0% to 100%), and evaluated the fuzzers on the generated benchmarks. Note that we used the default parameters (§5.2) to generate the benchmark, including the five default values of C_{seed} for each equality check proportion to produce a benchmark of 25 programs. Table 3 presents the results. In summary, as the proportion of equality checks increased from 0% to 100%, the code coverage achieved by most tools declined considerably, except Fuzzolic.

Table 3: Performance comparison with varying proportion of equality checks.

Measure	Tool	0%	25%	50%	75%	100%
Coverage (%)	AFL	**57.7**	28.7	22.1	20.9	16.6
	AFLGo	**52.2**	29.6	24.3	20.7	18.5
	AFL++	**87.7**	53.2	41.4	41.0	35.3
	Eclipser	**82.2**	28.7	19.2	18.0	13.4
	Fuzzolic	**86.6**	77.8	63.8	53.3	48.1
Bugs (%)	AFL	**64**	0	0	0	0
	AFLGo	**32**	0	0	0	0
	AFL++	**100**	16	36	12	20
	Eclipser	**100**	0	0	0	0
	Fuzzolic	**100**	88	80	40	28
TTE (h)	AFL	**18.20**	-	-	-	-
	AFLGo	**20.21**	-	-	-	-
	AFL++	**0.51**	18.59	12.75	13.85	18.09
	Eclipser	**1.50**	-	-	-	-
	Fuzzolic	**1.10**	8.10	11.83	15.54	17.86

The best results in each column and in each row are in grey and bold, respectively.

The results are intuitive because Fuzzolic uses symbolic execution which excels at finding solutions for equality constraints. Consequently, only Fuzzolic, and AFL++ to some extent, were able to find the bugs by penetrating all the synthesized conditions. Therefore, we conclude that equality constraints can still pose difficulties for many grey-box fuzzers.

5.5.2 Impact of Real-Constraints-based Rendering. Recall that Fuzzle can render all the conditional expressions solely based on the path formulas from C_{smt} (§4.3.2). We ran fuzzers on the benchmark generated with this strategy, and measured the performance. Table 4 compares the two results with or without using the strategy. With the real-constraints-based rendering strategy, the fuzzers achieved 24.32% less branch coverage, 9.87% less success rate, and 34.60%

Table 4: Performance comparison with or without real-constraints-based rendering strategy.

Measure	Tool	CVE-2016-4487		CVE-2016-4491		CVE-2016-4493	
		Range	Constr.	Range	Constr.	Range	Constr.
Coverage (%)	**AFL**	**37.8**	26.8	**25.3**	18.9	**24.6**	12.7
	AFLGo	**39.9**	30.5	**32.8**	21.3	**36.2**	16.3
	AFL++	75.8	**75.9**	**29.0**	26.2	**76.1**	51.0
	Eclipser	**74.2**	51.2	**25.4**	20.3	**61.1**	26.5
	Fuzzolic	79.8	**80.0**	55.4	**60.6**	**77.0**	70.6
Bugs (%)	**AFL**	**12**	4	0	0	0	0
	AFLGo	**8**	0	0	0	0	0
	AFL++	84	**100**	**4**	0	**80**	**80**
	Eclipser	**92**	72	0	0	**80**	8
	Fuzzolic	**100**	**100**	48	**84**	**100**	**100**
TTE (h)	**AFL**	21.47	**16.93**	-	-	-	-
	AFLGo	**19.72**	-	-	-	-	-
	AFL++	**2.43**	3.39	**18.73**	-	**6.26**	6.28
	Eclipser	**6.74**	8.95	-	-	**10.01**	15.12
	Fuzzolic	**1.59**	2.24	**7.73**	10.87	**2.41**	4.64

Table 5: Performance comparison with varying maze sizes.

		CVE-2016-4487			CVE-2016-4491			CVE-2016-4493		
		10×10	20×20	30×30	10×10	20×20	30×30	10×10	20×20	30×30
Cov. (%)	**AFL**	**89.1**	77.8	37.8	**62.9**	47.9	25.3	**86.3**	60.5	24.6
	AFLGo	**89.2**	78.1	39.9	**63.8**	58.6	32.8	**86.1**	61.7	36.2
	AFL++	**89.1**	87.6	75.8	**57.9**	50.1	29.0	**86.4**	81.6	76.1
	Eclipser	**89.2**	87.5	74.2	**65.8**	46.4	25.4	**84.7**	75.4	61.1
	Fuzzolic	**89.1**	84.0	79.8	80.6	**83.6**	55.4	**88.7**	83.7	77.0
Bugs (%)	**AFL**	**100**	76	12	8	**12**	0	**92**	36	0
	AFLGo	**100**	72	8	**24**	4	0	**92**	36	0
	AFL++	**100**	**100**	84	**8**	4	4	**96**	92	80
	Eclipser	**100**	**100**	92	**20**	0	0	**92**	84	80
	Fuzzolic	**100**	**100**	**100**	80	**96**	48	**100**	**100**	**100**
TTE (h)	**AFL**	**1.76**	15.08	21.47	**15.26**	15.27	-	**4.74**	16.90	-
	AFLGo	**1.88**	15.70	19.72	**13.18**	19.91	-	**5.10**	17.40	-
	AFL++	1.38	**1.32**	2.43	**17.14**	18.80	18.73	3.44	**3.35**	6.26
	Eclipser	**0.91**	4.25	6.74	**19.91**	-	-	**7.15**	9.62	10.01
	Fuzzolic	**0.38**	1.33	1.59	8.02	**6.76**	7.73	**1.23**	2.35	2.41

slower in finding bugs. This is because the generated programs include more complex conditionals compared to simple range checks. Thus, we conclude that the way of rendering path conditions can greatly impact the difficulty level of synthesized benchmarks.

5.6 RQ4: Customizable Difficulty

Recall from §5.4.4 C_{smt} significantly affects the performance of the fuzzers by making the synthesized bugs too difficult to find. Thus, a natural question follows: Can Fuzzle customize the difficulty of generated benchmark while still using the same C_{smt}? To answer this question, we generated programs with smaller sizes (10x10 and 20x20) while using the same C_{smt}. For each size, we generated five different programs using five PRNG seeds (C_{seed}). We then ran the fuzzers on all programs we generated (five times per program as described in §5.1.2). Table 5 summarizes the results with three different values for C_{smt} and C_{size}. Note the columns with C_{size} = 30x30 represent the results from the default setup used in §5.4.

Overall, the fuzzers found more bugs as we decrease C_{size}, demonstrating that the difficulty of finding the bug can easily be controlled with C_{size}, even if C_{smt} is used. For example, in the programs generated with CVE-2016-4493, both AFL and AFLGo were not able to find any bugs in the default-sized mazes, but found almost all bugs in 10x10 mazes. Furthermore, the fuzzers were able to find the bugs much quicker in smaller mazes. The time taken to find bugs in the programs generated with CVE-2016-4487 were reduced by 12x, 10x, 7x, 4x, for AFL, AFLGo, Eclipser, and Fuzzolic respectively. Thus, we conclude that varying C_{size} is an effective way to control the difficulty of generated benchmarks.

Table 6: Time to expose bugs in two different setups.

	AFL [62]		AFL++ [21]		Eclipser [14]		Fuzzolic [6]	
	Ours*	Orig†	Ours*	Orig†	Ours*	Orig†	Ours*	Orig†
CVE-2016-4487	21.7	3.8	1.6	**0.4**	5.5	0.9	**1.5**	1.3
CVE-2016-4489	20.7	8.8	5.9	**1.1**	7.9	1.6	**2.6**	2.7
CVE-2016-4491	> 24.0	> 24.0	18.7	8.9	> 24.0	> 24.0	**7.2**	**8.7**
CVE-2016-4492	23.5	7.3	**1.1**	**4.6**	4.3	6.9	1.3	7.6
CVE-2016-4493	> 24.0	4.6	4.1	**1.7**	7.5	4.3	**2.2**	1.9
CVE-2016-6131	> 24.0	> 24.0	9.9	> 24.0	11.9	> 24.0	**4.1**	> 24.0

* Fuzzle-generated benchmark. † GNU Binutils benchmark.

5.7 RQ5: Representative Performance

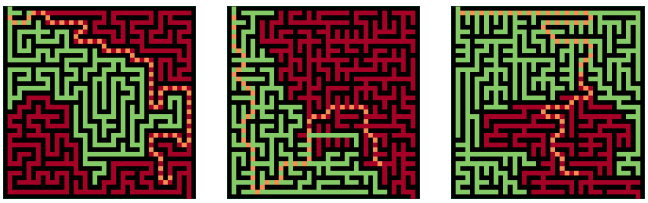
How do Fuzzle-generated programs compare to real-world programs? To answer this question, we ran the same set of fuzzers, excluding AFLGo which requires the source code to run, on both the Fuzzle-generated programs and the programs in GNU Binutils v2.26. When we synthesize programs, we used the default configuration except that we set C_{smt} with the six path formulas obtained by triggering the six CVEs from the same version of the Binutils programs. Table 6 shows the results. The "Ours" and "Orig" columns represent the time taken to find the bug on the synthesized programs and the original programs, respectively. Note that the reported numbers are the median of all 25 experimental runs.

Overall, both program sets show a similar trend in the difficulty level of finding the bugs. For example, most fuzzers found CVE-2016-4487 from Binutils within the first few hours of fuzzing, but struggled to find CVE-2016-4491 and CVE-2016-6131. The same trend is shown in the respective Fuzzle-generated programs. Note, however, that the fuzzers spent a slightly longer time to find the bugs in the synthesized programs compared to the original programs. One notable exception was AFL, which showed the worst performance overall. This is because it was spending most of its time exploring irrelevant maze paths added by Fuzzle.

Furthermore, Fuzzolic and AFL++ outperformed other fuzzers on both program sets. For example, in the programs generated with CVE-2016-4491, only Fuzzolic and AFL++ found the synthesized bug within 24 hours. Likewise, the two fuzzers are the only fuzzers that found CVE-2016-4491 in the original Binutils. Therefore the performance of fuzzers on the programs generated with C_{smt} are representative of their performance on real benchmarks.

5.8 RQ6: Coverage Visualization

Recall from §2, one of our motivations was to provide visual feedback regarding the progress of a fuzzing campaign. In this section,



(a) Backtracking (b) Prim's (c) Sidewinder

Figure 5: Coverage achieved by AFL for three different programs synthesized with three different algorithms.

we demonstrate that the coverage visualization is indeed useful in understanding and comparing the fuzzers.

5.8.1 Case Study 1. We first generated programs with three different maze generation algorithms (the backtracking, Prim's, and Sidewinder) and the path formula obtained from CVE-2016-4491, while using the default values for the other parameters. We then ran AFL once for 24 hours for each of the programs. The branch coverage achieved by the backtracking, Prim's, and Sidewinder were 31.2%, 42.4%, and 60.2%, respectively. We then converted the coverage results into the maze figures shown in Figure 5. Note the orange markings indicate that the marked edges are guarded with conditional expressions taken from C_{smt}. These markings, therefore, essentially highlight the buggy path of the program as we used the default rendering strategy. From these figures, we can easily see which regions were covered by AFL, and how close AFL was to finding the bug in the program.

5.8.2 Case Study 2. The visualization feature of Fuzzle can also be used to show the progress of fuzzing over time. To demonstrate our idea, we created an animated GIF showing the hourly code coverage improvement on a program in our benchmark. We provide a URL for reference.[2] The animated GIF clearly and intuitively shows the progress of the fuzzer. This idea can also be used to present real-time fuzzing status during a fuzzing campaign.

6 RELATED WORK

6.1 Bug Synthesis and Fuzzing Benchmarks

Current bug synthesizers are mainly inspired by mutation testing [9, 34, 43, 55], which aims to judge the quality of a test suite by mutating the programs. The key difference between the two is the evaluation target: bug synthesizers attempt to evaluate fuzzers, whereas mutation testers evaluate test suites.

LAVA [19, 20] is a seminal bug synthesizer that identifies potential attack points through dynamic taint analysis. For each identified point, it injects a bug that is guarded by a magic value check. EvilCoder [56] modifies the source code of the target program to remove protection mechanisms, such as input sanitization statements, and reintroduce the corresponding potential vulnerability that was previously guarded by those mechanisms. Apocalypse [58] focuses on generating challenging bugs for fuzzers by embedding a state machine such that the injected bug is only triggered when a series of non-trivial control-flow and dataflow conditions are met. Bug-Injector [36] inserts bugs based on bug templates into real-world programs to generate benchmarks for the evaluation of static analysis tools. It first finds a suitable injection location, where the state satisfies the preconditions for some bug templates, then it modifies the program to inject the buggy code. FixReverter [63] injects bugs by identifying code sites that match the bugfix patterns and reverting the fix. To ensure that the identified code site satisfies the semantic conditions for a triggerable bug, FixReverter performs static analysis and checks that the injection site is reachable and that there is dataflow from the variable involved in the bugfix pattern to a subsequent dereference site.

All these approaches introduce synthetic bugs by modifying a given program. Note, however, Fuzzle is unique in that it synthesizes a buggy program from scratch. Indeed, our technique is closely related to program synthesis [24, 33, 45, 51, 53]. There is a recent fuzzing benchmark, named Magma [26], which is orthogonal to our work as the benchmark is manually generated by careful auditing.

6.2 Maze and Software Testing

Mazes have been commonly used for evaluating symbolic executors, such as KLEE. One famous example of such maze programs is demonstrated in Felipe's tutorial [49]. Fuzzle, inspired by Felipe's maze programs, also synthesizes programs based on mazes. The two approaches, nevertheless, differ in two major ways. First, Felipe's approach uses a two-dimensional array and its indices to represent the maze and the cells in the maze, respectively. However, in Fuzzle, each cell of the maze is represented as a function in the program, which renders the shape of the resulting program similar to it of recursive descent parsers. The maze program generated by Fuzzle is unique also in that it provides only the valid choices at each cell, instead of always allowing all 4 directions as in the Felipe's implementation. Fuzzle's approach more closely resembles the intuitive way of solving mazes, as it prevents the immediate termination of the program upon choosing one of the invalid choices, i.e. crashing into the wall of the maze.

6.3 Fuzzing

Fuzz testing (or fuzzing) has been a great success in finding various software bugs [2, 8, 12, 15, 16, 25, 39–42, 47, 48, 50, 57]. In this paper, we use the term "fuzzing" to mean various different testing techniques that involve automatic test case generation, such as concolic testing [7, 23], grey-box fuzzing [3, 14], and hybrid fuzzing [6]. Although there have been remarkable advances in the field, there still lack large-scale benchmarks that provide precise ground truth. Fuzzle takes the first step toward synthesizing a fuzzing benchmark without modifying the existing programs.

7 DISCUSSION AND FUTURE WORK

We are aware that, as in any empirical study, there are various threats to the validity of our results and conclusions. To mitigate the threat of the selection bias, we chose fuzzers of various types, covering grey-box fuzzing, symbolic execution, and directed fuzzing. Furthermore, we generated a large benchmark for fuzzer evaluation by varying each parameter values to ensure that our results are not restricted to a particular setup of parameter values. The evaluation of Fuzzle's visualization feature is also subject to experimenter

[2] https://softsec-kaist.github.io/Fuzzle

bias, as the usefulness of the feature was assessed based on our interpretation of the results. We leave it as future work to perform a user study to further validate our visualization feature.

As Fuzzle generates buggy programs from mazes, the fuzzers may try to adapt to the system, simply by using a maze solver to find the solution path of the maze prior to fuzzing. Nevertheless, our design is not fundamentally limited to such an overfitting problem as we allow any node in a maze to be a terminal node (i.e., the buggy function). Additionally, we can obfuscate the buggy function so that it cannot be easily detected. Specifically, we can add an abort system call that is guarded with an opaque predicate [17] in every function of the program. In this way, every program generated by Fuzzle contains a randomly placed bug, preventing fuzzers from taking advantage of the maze-based design of the programs.

Distinguishing between buggy and non-buggy paths is fundamentally difficult when Fuzzle uses the real-constraints-based approach (§4.3.2), because it renders every conditional using real path constraints. Additionally, we can employ different strategies to synthesize the conditionals. For example, we can randomly synthesize our branch conditions similar to the way Csmith [61] generates random C programs. The programs generated in this way would be more predictable than the human-developed programs comparable to how the bug-triggering Csmith fuzzer-generated programs are highly repetitive and predictable [35].

While Fuzzle can encode various real-world bug types by considering buggy path conditions, it cannot synthesize bugs that involve subtle data-flow changes. For example, it currently cannot synthesize an off-by-one error. One may expand the current design of Fuzzle to enable data flows between functions, and we believe this is a promising direction for future research.

We can further extend Fuzzle by employing a global variable to fine-control path conditions and introduce complex program states (§4.2). For example, we can introduce a counter variable to keep track of the number of times each function has been called. Then the value of such variable can be used in the conditions required to invoke the function calls or to terminate the program when it reaches a predefined limit. Lastly, we can allow generated programs to have more than 4 edges per node by generating mazes of different shapes, such as those on a hexagonal grid. We believe extending Fuzzle to support these features is a promising future work.

8 CONCLUSION

In this paper, we introduced Fuzzle, the first bug synthesizer that does not rely on existing programs. The key intuition of Fuzzle is to convert every path in a randomly generated maze into a call path of the synthesized program. Our design provides intuitive visual feedback for software testers, while enabling fine-grained control on the buggy logic. We evaluated Fuzzle with five state-of-the-art fuzzers to understand its value. We empirically showed that Fuzzle is an efficient bug synthesizer that can effectively evaluate various state-of-the-art fuzzers.

ACKNOWLEDGMENTS

We thank anonymous reviewers for their constructive feedback. This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2021-0-01332, Developing Next-Generation Binary Decompiler).

REFERENCES

[1] Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the International Conference on Software Engineering*. 1–10.
[2] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. 2020. Boosting Fuzzer Efficiency: An Information Theoretic Perspective. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 678–689.
[3] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the ACM Conference on Computer and Communications Security*. 2329–2344.
[4] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the ACM Conference on Computer and Communications Security*. 1032–1043.
[5] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. 2021. Fuzzing Symbolic Expressions. In *Proceedings of the International Conference on Software Engineering*. 711–722.
[6] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. 2021. Fuzzolic: Mixing Fuzzing and Concolic Execution. *Computers & Security* 108 (2021).
[7] Ella Bounimova, Patrice Godefroid, and David Molnar. 2013. Billions and Billions of Constraints: Whitebox Fuzz Testing in Production. In *Proceedings of the International Conference on Software Engineering*. 122–131.
[8] Tegan Brennan, Seemanta Saha, and Tevfik Bultan. 2020. JVM Fuzzing for JIT-Induced Side-Channel Detection. In *Proceedings of the International Conference on Software Engineering*. 1011–1023.
[9] David Bingham Brown, Michael Vaughn, Ben Liblit, and Thomas Reps. 2017. The Care and Feeding of Wild-Caught Mutants. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 511–522.
[10] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation*. 209–224.
[11] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *Proceedings of the IEEE Symposium on Security and Privacy*. 380–394.
[12] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-Adaptive Mutational Fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy*. 725–741.
[13] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Grey-box Fuzzer. In *Proceedings of the ACM Conference on Computer and Communications Security*. 2095–2108.
[14] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. 2019. Grey-box Concolic Testing on Binary Code. In *Proceedings of the International Conference on Software Engineering*. 736–747.
[15] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. 2021. SMARTIAN: Enhancing Smart Contract Fuzzing with Static and Dynamic Data-Flow Analyses. In *Proceedings of the International Conference on Automated Software Engineering*. 227–239.
[16] Jaeseung Choi, Kangsu Kim, Daejin Lee, and Sang Kil Cha. 2021. NTFuzz: Enabling Type-Aware Kernel Fuzzing on Windows with Static Binary Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*. 1973–1989.
[17] Christian Collberg, Clark Thomborson, and Douglas Low. 1998. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. 184–196.
[18] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. 2009. *Introduction to Algorithms* (3 ed.). The MIT Press.
[19] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. LAVA: Large-scale Automated Vulnerability Addition. In *Proceedings of the IEEE Symposium on Security and Privacy*. 110–121.
[20] Andrew Fasano, Tim Leek, Brendan Dolan-Gavitt, and Josh Bundt. 2019. The Rode0day to Less-Buggy Programs. *IEEE Security and Privacy* 17, 6 (2019), 84–88.
[21] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. USENIX Workshop on Offensive Technologies.
[22] Marco Gario and Andrea Micheli. 2015. pySMT: A library for SMT formulae manipulation and solving. https://github.com/pysmt/pysmt.
[23] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Commun. ACM* 55, 3 (2012), 40–44.
[24] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. 317–330.
[25] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. 2018. DLFuzz: Differential Fuzzing Testing of Deep Learning Systems. In *Proceedings of the*

International Symposium on Foundations of Software Engineering. 739–743.
[26] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 3 (2020), 1–29.
[27] J. D. Hunter. 2007. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering* 9, 3 (2007), 90–95.
[28] Marnie L. Hutcheson. 2003. *Software Testing Fundamentals: Methods and Metrics* (1 ed.). Wiley.
[29] Free Software Foundation Inc. [n.d.]. GCC 3.4 Release Series—Changes, New Features, and Fixes. https://gcc.gnu.org/gcc-3.4/changes.html.
[30] Free Software Foundation Inc. [n.d.]. GCC 4.1 Release Series—Changes, New Features, and Fixes. https://gcc.gnu.org/gcc-4.1/changes.html.
[31] Free Software Foundation Inc. [n.d.]. GCC, the GNU Compiler Collection. https://gcc.gnu.org/.
[32] Google Inc. 2016. fuzzer-test-suite. https://github.com/google/fuzzer-test-suite.
[33] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-Guided Component-Based Program Synthesis. In *Proceedings of the International Conference on Software Engineering*. 215–224.
[34] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678.
[35] Rajeswari Hita Kambhamettu, John Billos, Tomi Oluwaseun-Apo, Benjamin Gafford, Rohan Padhye, and Vincent J. Hellendoorn. 2022. On the Naturalness of Fuzzer-Generated Code. In *Proceedings of the International Conference on Mining Software Repositories*.
[36] Vineeth Kashyap, Jason Ruchti, Lucja Kot, Emma Turetsky, Rebecca Swords, Shih An Pan, Julien Henry, David Melski, and Eric Schulte. 2019. Automated Customized Bug-Benchmark Generation. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation*. 103–114.
[37] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the ACM Conference on Computer and Communications Security*. 2123–2138.
[38] lafintel. 2016. Circumventing Fuzzing Roadblocks with Compiler Transformations. https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/.
[39] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: Automatically Generating Pathological Inputs. In *Proceedings of the International Symposium on Software Testing and Analysis*. 254–265.
[40] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-state Based Binary Fuzzing. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 627–637.
[41] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. 2019. Cerebro: Context-Aware Adaptive Fuzzing for Effective Vulnerability Detection. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 533–544.
[42] Daniel Liew, Cristian Cadar, Alastair F Donaldson, and J Ryan Stinnett. 2019. Just Fuzz It: Solving Floating-Point Constraints using Coverage-Guided Fuzzing. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 521–532.
[43] Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2017. Enabling Mutation Testing for Android Apps. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 233–244.
[44] Dongge Liu, Gidon Ernst, Toby Murray, and Benjamin I. P. Rubinstein. 2020. Legion: Best-First Concolic Testing. In *Proceedings of the International Conference on Automated Software Engineering*. 54–65.
[45] Fan Long and Martin Rinard. 2015. Staged Program Repair with Condition Synthesis. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 166–178.
[46] Chenyang Lyu, Shouling Ji, Xuhong Zhang, Hong Liang, Binbin Zhao, Kangjie Lu, and Raheem Beyah. 2022. EMS: History-Driven Mutation for Coverage-based Fuzzing. In *Proceedings of the Network and Distributed System Security Symposium*.
[47] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2312–2331.
[48] Valentin J. M. Manès, Soomin Kim, and Sang Kil Cha. 2020. Ankou: Guiding Grey-box Fuzzing towards Combinatorial Difference. In *Proceedings of the International Conference on Software Engineering*. 1024–1036.
[49] Felipe Andres Manzano. 2010. The Symbolic Maze! https://feliam.wordpress.com/2010/10/07/the-symbolic-maze/.
[50] Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Matthias Höschele, and Andreas Zeller. 2019. Parser-directed Fuzzing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 548–560.
[51] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the International Conference on Software Engineering*. 691–701.
[52] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 1393–1403.
[53] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the International Conference on Software Engineering*. 772–781.
[54] NIST. 2016. CVE-2016-6131. https://nvd.nist.gov/vuln/detail/CVE-2016-6131.
[55] Jibesh Patra and Michael Pradel. 2021. Semantic Bug Seeding: A Learning-Based Approach for Creating Realistic Bugs. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 906–918.
[56] Jannik Pewny and Thorsten Holz. 2016. EvilCoder: Automated Bug Insertion. In *Proceedings of the Annual Computer Security Applications Conference*. 214–225.
[57] Siegfried Rasthofer, Steven Arzt, Stefan Triller, and Michael Pradel. 2017. Making Malory Behave Maliciously: Targeted Fuzzing of Android Execution Environments. In *Proceedings of the International Conference on Software Engineering*. 300–311.
[58] Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. 2018. Bug Synthesis: Challenging Bug-Finding Tools with Deep Faults. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 224–234.
[59] Shivani H. Shah, Jagruti M. Mohite, Anoop G. Musale, and Jay L. Borade. 2017. Survey Paper on Maze Generation Algorithms for Puzzle Solving Games. *International Journal of Scientific and Engineering Research* 8, 2 (2017), 1064–1067.
[60] John Stilley. 2014. mazelib: A Python API to generate and solve mazes. https://github.com/john-science/mazelib.
[61] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. 283–294.
[62] Michal Zalewski. [n.d.]. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.
[63] Zenong Zhang, Zach Patterson, Michael Hicks, and Shiyi Wei. 2022. FixReverter: A Realistic Bug Injection Methodology for Benchmarking Fuzz Testing. In *Proceedings of the USENIX Security Symposium*. 3699–3715.