Program Feature-based Benchmarking for Fuzz Testing

ANONYMOUS AUTHOR(S)

Fuzzing is a powerful software testing technique renowned for its effectiveness in identifying software vulnerabilities. Traditional fuzzing evaluations typically focus on overall fuzzer performance across a set of target programs, yet few benchmarks consider how fine-grained program features influence fuzzing effectiveness. To bridge this gap, we introduce `FeatureBench`, a novel benchmark designed to generate programs with configurable, fine-grained program features to enhance fuzzing evaluations. We reviewed 25 recent grey-box fuzzing studies, extracting 7 program features related to control-flow and data-flow that can impact fuzzer performance. Using these features, we generated a benchmark consisting of 153 programs controlled by 10 fine-grained configurable parameters. We evaluated 11 fuzzers using this benchmark, with each fuzzer representing either distinct claimed improvements or serving as a widely used baseline in fuzzing evaluations. The results indicate that fuzzer performance varies significantly based on the program features and their strengths, highlighting the importance of incorporating program characteristics into fuzzing evaluations.

ACM Reference Format:
Anonymous Author(s). 2025. Program Feature-based Benchmarking for Fuzz Testing. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '25).* ACM, New York, NY, USA, 22 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Fuzzing, or fuzz testing, is a powerful software testing technique renowned for its effectiveness in software and system security testing. Numerous fuzzers have been proposed in recent years, each improving different components of a given fuzzer. For example, AFLFast [4] introduces new search strategies and power schedules designed to prioritize seeds that traverse less frequently executed paths. EcoFuzz [53] claims to implement an adaptive power schedule that reduces energy wastage and maximizes path coverage within a finite execution time.

The evaluation of fuzzers is usually conducted on a set of target programs, focusing on the overall performance (e.g., bug finding capability and coverage after the preset timeout), compared to a set of baselines. We observe that such evaluations often reveal that different fuzzers tend to favor specific programs. For instance, fuzzers' performance often varies across different target programs in the evaluations that use FuzzBench (which uses code coverage to rank different fuzzers) [34]. One of the reasons for such variation lies in the design of any given fuzzer. For example, EcoFuzz's advantage in reducing energy wastage and maximizing path coverage [53] may become more pronounced as program complexity increases. However, current evaluations do not consider program features or analyze performance deviations in relation to those features. Therefore, the research community has yet to establish a link between fuzzing performance and program features; without this link, it remains unknown if the hypotheses and claims made in these fuzzers hold, making it hard to assess and further improve them.

To close this gap, we propose to develop the first program feature-based benchmark for fuzzing. The program features we considered are extracted from the literatures that propose improvements to fuzzers. These features describe the control-flow and data-flow complexity of target programs. For example, features like *number of conditional branches* and *execution probability of conditional branches* affect the control-flow complexity of a target program and determine how likely it is for a branch to be executed. Similarly, the depth of *nested magic bytes and checksum tests* or the length of *magic bytes* contribute to the data-flow complexity of a target program, increasing the difficulty for fuzzers to satisfy these constraints and reach buggy code. Evaluating fuzzers on such set of programs can offer deeper insights into how their performance varies on different program features and offer comparison of fuzzing performance across various configurations of the same program feature. The programs in our benchmark are synthetically generated to provide a controlled environment for fuzzers and allow us to better understand the impact of these features on the performance of different fuzzers, thereby improving explainability of the observed fuzzing behaviors. Our benchmark also offers fine-grained configurable parameters to control the

ISSTA '25, June 25–28, 2025, Trondheim, Norway

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

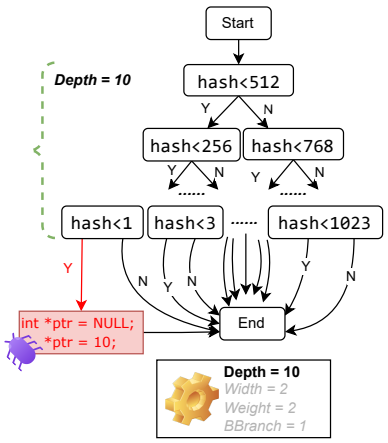
construction of the synthetic programs that represent specific features related to the program's control-flow and data-flow complexity. For example, for fuzzers that optimize efficiency through smart seed selection, mutation strategies, or energy allocation, increasing the number of branches or the maximum depth of nested constraints within the program shall allow us to test their ability to scale and handle more complex programs.

To develop such feature-based benchmark, we first reviewed 25 existing grey-box fuzzing papers and their implementations to summarize their claimed improvements. We then extracted program features that describe the *control-flow complexity* and *data-flow complexity* of target programs. For example, a higher number of conditional branches increases the number of possible execution paths within a program. As a result, fuzzers will likely need to traverse more paths to reach the buggy code, thereby increasing the difficulty of generating bug-triggering inputs. We extracted this feature from the literature that propose enhancements to improve overall fuzzing efficiency through effective strategies, such as energy allocation and mutation algorithms. These fuzzers can maximize path coverage within a limited time and number of executions, and tackle the challenges posed by complex control flows. In total, we extracted 7 program features from the literature, including *number of conditional branches*, *execution probability of conditional branches*, *loops and recursions*, *data-constrained loops and recursions*, *magic bytes*, *checksum tests*, and *nested magic bytes and checksum tests*.

We then designed and generated benchmarks based on these extracted programs features. We created multiple configurable parameters to control the complexity of the programs in a fine-grained manner for each feature and discussed the selection of experimental settings for each parameter in detail. For example, four parameters (`Width`, `Depth`, `Weight`, `BBranch`) are created for the control-flow complexity features. `Width` defines the number of branching paths from each `if` condition. `Depth` determines the nesting level of conditional statements. `Weight` controls the probability of each conditional branch being executed, and `BBranch` determines on which branch the bug is located. Figure 1(a) illustrates the skeleton of the programs with the default settings of `Width` (2), `Weight` (2), `BBranch` (1), and a variable `Depth` that can be configured to any positive integer. The branching paths from each `if` condition in these programs have the same probability to be executed, and the bug always locates on the first branch. Figure 1(b) demonstrates a control-flow graph of a program generated based on the skeleton in Figure 1(a), with the `Depth` set to 10 (see detailed discussion in Section 4.1). In total, we generated 153 programs controlled by 10 parameters, targeting the 7 distinct program features.

```
void COMP_W2_D$Depth_ω2_B1(unsigned hash)
{if (hash < pow(2, $Depth-1})) {// level1
    if (hash < pow(2, $Depth-2)) {// level2
       ...
          if (hash < 1) {// level$Depth
              PRINTF("This is branch 1\n");
              /* Insert a bug here */
          } else {// level$Depth
              PRINTF("This is branch 2\n");
          }
       ...
 } else {// level1
    if (hash < pow(2, $Depth)-pow(2, $Depth-2)) {// level2
       ...
    else {// level2
       if (hash < pow(2, $Depth)-pow(2, $Depth-3)) {// level3
          ...
          if (hash < pow(2, $Depth)-pow(2, 0)) {// level$Depth
              PRINTF("This is branch {pow(2, $Depth)-pow(2, 0)}\n");
          ...
 }
```

(a) Program skeleton.



(b) Control-flow graph.

Fig. 1. Illustrative example of control-flow complexity parameters.

We evaluated 11 fuzzers using our feature-based benchmark, `FeatureBench`. We selected fuzzers that represent the improvements from which the program features implemented in `FeatureBench` are extracted, and included popular coverage-guided fuzzers from FuzzBench [35], such as AFL [54], AFL++ [10], and Honggfuzz [13]. These fuzzers are frequently used as baselines in fuzzing evaluations [27, 34, 53, 55, 59]. Note that while these fuzzers have been used in previous evaluations, these evaluations did not consider the impact of program features, like those we have incorporated into our benchmark suite. With `FeatureBench`, we perform correlation analysis and use data visualization to understand the impact of each program parameter on the performance of different

fuzzers and reports the results on how well each feature is supported by the evaluated fuzzers. Our findings show that fuzzer performance varies significantly depending on program features and the strength of those features. For example, RedQueen [3] and Laf-intel [22] perform well on programs with high `Depth` but struggle with high `Width`. Memlock(stack) [51], TortoiseFuzz [50] and AFL++ [10], although struggling with the long magic bytes, are able to find bugs guarded by deeply nested magic bytes. These findings highlight the importance of considering program characteristics in fuzzing evaluations. We made the following contributions in this work:

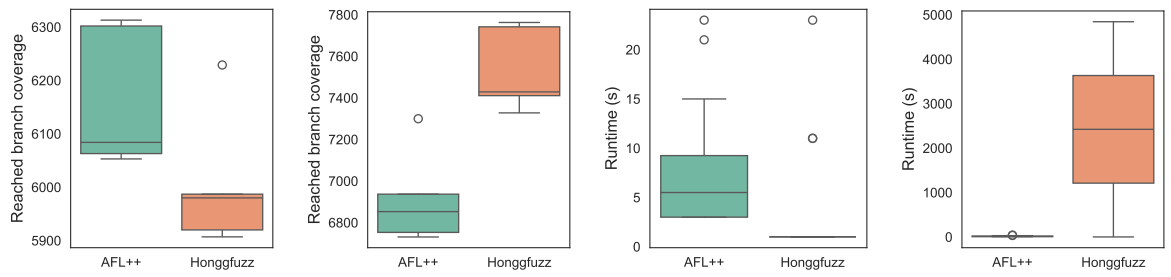
- We perform a literature review of 25 recent grey-box fuzzing papers to extract 7 fine-grained program features from their claimed improvements.
- We create a novel feature-based benchmark, `FeatureBench`, for evaluating fuzzers, that defines 10 configurable parameters for the extracted program features with 153 generated programs.
- We evaluate 11 popular fuzzers on `FeatureBench` to understand fuzzer behaviors and the impact of each program parameter on their performance.

2 Background and Motivation

2.1 Motivating Example and Experiment

It has been a prevalent observation that different fuzzers seem to favor different target programs as their ability to find bugs or achieve high code coverage varies across different programs. In a sample report from FuzzBench [12] that shows the ranking results of 11 different fuzzers on 20 different target programs based on code coverage, Honggfuzz [13], LibFuzzer [30], AFL++ [10], and MOpt [31] all have been ranked first on at least one target program. UNIFUZZ [27] also observed that no evaluated fuzzer consistently outperforms the others across all target programs in terms of unique bugs found. However, currently there is a lack of analyses summarizing the characteristics of these target programs to explain the differences in fuzzing performance.

We replicated part of the experiments in the sample report from FuzzBench [12]. Based on the ranking results in the report, AFL++ [10] and Honggfuzz [13] exhibited highly inconsistent rankings across different target programs. We ran these two fuzzers on two target programs (bloaty_fuzz_target and proj4_proj_crs_to_crs) in FuzzBench, executing each fuzzer on each target program for 24 hours across five iterations. Figures 2(a) and 2(b) illustrate the results of this experiment. FuzzBench ranks fuzzers based on the median coverage they reached within 24 hours, and we observe that AFL++ outperformed Honggfuzz on bloaty_fuzz_target, whereas Honggfuzz outperformed AFL++ on proj4_proj_crs_to_crs. However, FuzzBench does not report the reason of this performance inconsistency; for example, the program features that may have caused this inconsistency were not analyzed by FuzzBench.



(a) bloaty_fuzz_target. (b) proj4_proj_crs_to_crs. (c) COMP_W2_D12_ω2_B1. (d) COMP_W2_D16_ω2_B1.

Fig. 2. FuzzBench (left two) vs. crafted program (right two) results on AFL++ and Honggfuzz.

Indeed, the design of a fuzzer may actually cause its varying performance across different programs. To better understand the reason behind the performance differences, we assume that the control-flow complexity of the target program might be one of the factors influencing the fuzzing performance. Based on this assumption, we performed an experiment to compare AFL++ and Honggfuzz on two manually crafted programs, `COMP_W2_D12_ω2_B1` and `COMP_W2_D16_ω2_B1`. Figure 1 illustrates the skeleton of these crafted programs, which are designed to have a control-flow graph that is a fully balanced tree with `Width` of 2, a varying `Depth`, an equal probability of every branch being traversed, and the bug located on the first branch. The two programs used in this experiment are generated based on this skeleton, but with different `Depth` (12 and 16). More details of these parameters will be explained in Section 4.

In our experiment, we set both fuzzers to stop fuzzing once a bug is found, and compare the running time it takes to find the bug. We fuzzed each program 20 times and calculated the average runtime to account for randomness during fuzzing. Figures 2(c) and 2(d) show the results of this experiment. We observe that on the smaller program (`COMP_W2_D12_ω2_B1`), Hongfuzz has a better performance than AFL++, almost immediately finding the bug, while AFL++ takes around 6 seconds to find the bug. However, as the programs grow more complicated in depth, AFL++ starts to outperform Honggfuzz, with Honggfuzz clearly taking longer time to find the bug on the larger programs (on average 2423 seconds for the program with the depth of 16), while AFL++'s runtime remains relatively low (on average 6-15 seconds) in both program variants. This observation confirms our assumption that control-flow complexity, specifically the `Depth` of the target program, is one of the factors influencing the Honggfuzz's performance. This promising result motivates us to explore relevant program features and design benchmarking programs that help in better understanding the performance differences of fuzzers across various program features.

2.2 Background

There exist several important fuzzing benchmarks. FuzzBench [35] provides an infrastructure to evaluate fuzzers in terms of code coverage and vulnerability exposure. It uses benchmark applications from OSS-Fuzz [46], which include real-world projects. Magma [14] provides a benchmark of 138 ported bugs in 9 open source programs along with the lightweight oracle (ground truth) that reports the bug when triggered. FixReverter [57] focuses on realism of bugs and aims at re-introducing a bug that was fixed before. LAVA-M [8] injects an out-of-bounds access that is guarded by a "magic value" comparison whereas CGC [6] contains small synthetic bugs: one bug per program. These benchmarks tend to focus on bugs or vulnerabilities in programs but do not explain performance differences across fuzzers on different target programs. As shown in our motivating experiment, a benchmark such as FuzzBench may rank fuzzers based on the median coverage they reached within 24 hours on each target program but the reason behind the ranking differences across different target programs remains unexplained. Our benchmark aims to complement these benchmarks and fill this gap by evaluating fuzzers on a set of benchmark programs with different features and analyzing how the fuzzing performance is influenced by these features.

UNIFUZZ [27] proposes a collection of pragmatic performance metrics to evaluate fuzzers from six complementary perspectives, as the authors believe using a single metric to assess the performance of a fuzzer may lead to unilateral conclusions. GreenBench [36] focuses on energy consumption of fuzzing evaluations. It creates thousands of benchmarks by using the existing FuzzBench programs with diverse seed inputs and runs on these benchmarks for a short period of time (i.e., minutes), and still generates accurate performance ranking results. Although these approaches bring new dimensions to understand fuzzer performance, they still overlook the influence of specific characteristics in target programs that can impact fuzzer effectiveness.

3 Program Feature Extraction

We reviewed 25 grey-box fuzzing papers that are published within last three years, as well as the most cited fuzzers from earlier years, and summarized the common hypotheses or claims of improvements on fuzzing performance. It is not our goal to cover all published fuzzing papers. Instead, we reviewed these popular fuzzing papers as a representative set to extract important program features to construct the benchmark. A benchmark based on these features can be used to validate fuzzers' claimed improvements and to understand performance differences of fuzzers across various program features. We extracted 7 program features from two aspects: *control-flow complexity* and *data-flow complexity*.

3.1 Control-Flow Complexity

We reviewed 15 papers that discuss improvements associated with control-flow complexity of target programs. Control-flow complexity in this context can be defined by four program features: *number of conditional branches*, *execution probability of conditional branches*, *loops and recursions* and *loops and recursions with data constraints*.

Number of conditional branches. We abstract a program as a control-flow graph, where non-leaf nodes represent `if`-condition checks and outgoing edges represent the possible branches. When a program has more conditional branches, the fuzzers will likely need to traverse more paths to reach the buggy code. This creates a challenge for the fuzzers to generate bug-triggering inputs. Papers that propose enhancements to improve overall fuzzing efficiency include effective strategies to handle the challenges posed by this program feature. For example, fuzzers like EcoFuzz [53], MooFuzz [59], MobFuzz [55], and Slime [32] seek to improve fuzzing efficiency through smart energy allocation. EcoFuzz claims that it "implements an adaptive schedule that can effectively reduce energy wastage, maximising path coverage within a limited number of executions" [53]. MooFuzz claims that

"reasonable energy allocation can effectively improve the discovery of new paths"[59]. Other fuzzers like MOpt [31], Darwin [17], ShapFuzz [56], FairFuzz [25], SEAMFuzz [24], and MobFuzz [55] propose improvements that optimize mutators to generate interesting test cases, that can trigger new paths or crashes more efficiently. One would thus expect fuzzers with smart energy allocation and/or mutators to outperform those without such advancements, especially in large programs where more conditional branches can be executed. One would anticipate that such advantage shall be more pronounced as the complexity of the program increases.

Execution probability of conditional branches. The execution probability of conditional branches refers to the likelihood of each branch being executed. When a bug resides in a hard-to-reach region of the code, the low likelihood of reaching that region poses significant challenges for fuzzers to generate inputs that can trigger the bug. Fuzzers that prioritize seeds that traverse infrequently executed paths seem more likely to effectively handle these challenges. For example, AFLFast [4], FairFuzz [25], DigFuzz [58], and rare path guided fuzzing [43] design special seed selection strategies and/or power schedules to increase the chances of discovering bugs in the hard-to-reach regions of the program. AFLFast prioritizes seeds that traverse infrequently executed paths [4], while FairFuzz identifies program branches that are rarely hit by previously generated inputs and increases the likelihood of hitting these rare branches [25]. One would expect fuzzers that optimize to prioritize seeds executing hard-to-reach code regions will excel in programs where bugs are located in infrequently reached areas.

Loops and recursions. Apart from the horizontal and vertical complexity of the control flow graph, the presence of loops and recursions in a program can also be used to measure the control-flow complexity. TortoiseFuzz [50] claims that "loops are widely used for accessing data and are closely related to memory errors such as overflow vulnerabilities." It utilizes the presence of loops to guide the fuzzing process by only considering the security-sensitive edges when calculating coverage gain. PATA [28] implements path-aware taint analysis to distinguish between multiple occurrences of the same variable, such as in loops or at different function call sites. Memlock [51], on the other hand, studies an uncontrolled-recursion bug. The bug requires a sufficiently large recursive depth which can lead to excessive memory consumption to trigger a stack overflow crash. Memory consumption refers to the stack/heap memory usage of the target program when executing an input. Some traditional coverage-based grey-box fuzzers such as AFL, although are aware of repeatedly executed CFG edges in a coarse manner, do not have awareness about such memory-related information. Due to this limitation, AFL does not differentiate the change when the recursive depth is greater than 255 [51]. However, MemLock intentionally keeps seeds that increase the peak length of call stack, and can finally triggering the stack-overflow.

Loops and recursions with data constraints. In addition to loops and recursions as measures of control-flow complexity, incorporating data-flow complexity provides a more comprehensive approach for designing benchmarks that assess control-flow complexity. We extract another feature that measures the presence and depth of loops and recursions while also considering data constraints that must be satisfied by the inputs.

3.2 Data-flow Complexity

We reviewed 10 papers proposing improvements that are associated with the data-flow complexity of the target program. The complex data-flow conditions refer to hard-to-fulfil conditions along the execution paths that guard certain regions of code where bugs might be located. We extracted three program features that can represent the data-flow complexity of the target program: *magic bytes*, *checksum tests*, and *nested magic bytes and checksum tests*.

Magic bytes. Magic bytes are a sequence of bytes commonly used to validate the format of a file or protocol, ensuring that the data conforms to the expected structure. They are frequently seen in real-world programs, such as file parsers and network protocols. These constructs are hard to solve for feedback-driven fuzzers since they are very unlikely to guess a satisfying input. Extensive research has been conducted to tackle such challenges. For example, RedQueen [3], TensileFuzz [29], Angora [7], Steelix [26], Vuzzer [40], T-Fuzz [38], Pangolin [16], RNNFuzzer [39], and Laf-intel [22] all have proposed improvements focused on resolving magic bytes in the target programs. Angora [7], Steelix [26], Vuzzer [40], T-Fuzz [38], and Pangolin [16] employ techniques like taint tracking and symbolic or concolic execution to bypass these roadblocks. There are lightweight techniques, such as the *input-to-state correspondence* method proposed by RedQueen [3], the LLVM [23] passes implemented by Laf-intel [22], as well as learning-based approaches like RNNfuzzer [39]. These fuzzers have been designed to be particularly effective at resolving or bypassing magic bytes checks.

Checksum tests. Another type of data-flow complexity is checksum tests, which are often used in network programs to detect data corruption. Due to its significant complexity, a checksum test is another common challenge for fuzzers to efficiently fuzz beyond. RedQueen [3], T-Fuzz [38], and TaintScope [49] are examples of fuzzers that have proposed improvements to tackle checksum tests. TaintScope and T-Fuzz remove the checksum tests from

Table 1. `FeatureBench` features, parameters, settings, and number of programs.

Category	Feature	Parameter	Settings	# of Programs	Total
Control-Flow	Number of conditional branches	Width	{32,48,64...256}	16 (COMW)	103
		Depth	{2,4,6...16}	8 (COMD)	
	Execution probability of conditional branches	BBranch	{1,32,64...1024}	32 (COMB)	
		Weight	{2,3,4...8}	7 (COMWE)	
	Loops and recursions	Iteration	{5,10,50,100...100000}	10 (LOOPI)	
			{False}	10 (RECURI)	
	Loops and recursions with data constraints	Has_Data_Constraint	{50,100,150...500}	10 (LOOPDI)	
			{True}	10 (RECURDI)	
Data-Flow	Magic bytes	Start	{0,10,20...90}	10 (MAGICS)	50
		Length	{1,2,3...10}	10 (MAGICL)	
	Checksum tests	Count	{1,2,3...10}	10 (CHECKSUMC)	
	Nested magic bytes and checksum tests	Depth	{1,2,3...10}	10 (MAGICD)	
			{1,2,3...10}	10 (CHECKSUMD)	
Total					**153**

the target program and seek to fulfil them later. They detect critical checks automatically, and then use symbolic execution to fulfill the checks once interesting behavior was found, while RedQueen uses the *input-to-state correspondence* method to bypass checksum tests.

Nested magic bytes and checksum tests. Some bugs are protected by a chain of hard-to-fulfill checks, such as nested magic bytes and checksum tests. These bugs are particularly challenging to trigger because they are located deep within the code, and fuzzers must satisfy multiple conditions to reach the buggy code. Fuzzers such as RedQueen [3], Laf-intel [22], Angora [7], Vuzzer [40], and RNNFuzzer [39] are also designed to excel at resolving such nested conditions.

4 Benchmark Generation

To construct a program feature-based benchmark, we generate synthetic programs emphasizing specific features with varying levels of strength to assess fuzzer performance. We adjust control- and data-flow complexity by stacking template blocks, and use fine-grained configurable parameters to control each feature's strength. This allows us to observe the trend of fuzzer performance in relation to incremental changes in each program feature. Each program includes a single injected bug, allowing us to measure the time each fuzzer takes to trigger the bug. We did not choose to inject multiple bugs into each program, as our primary focus is evaluating the fuzzer's efficiency in reaching the buggy code on programs emphasizing different features. We believe that injecting one single bug is sufficient for this purpose. Overall, our generated programs offer fine-grained control over the control-flow and data-flow complexity in a way that no existing benchmark can provide. Table 1 shows an overview of our benchmark, `FeatureBench`, including the categories, features, parameters, settings, and the number of programs generated for each feature. The **Feature** column lists all program features for both control-flow and data-flow categories, while the **Parameter** column lists the 10 configurable parameters used to control the strength of each feature. The **Settings** column shows the settings for each parameter we used to generate the benchmark. The **# of Programs** column shows the number of programs generated for each parameter, along with the corresponding program group name in parentheses. These programs are available in the respective folders of our benchmark [1]. In summary, we crafted 10 configurable parameters and generated a total of 153 programs, targeting 7 distinct program features.

4.1 Control-Flow Complexity

To manipulate the control-flow complexity of the programs with a finer granularity, we define six parameters. The first four, `Width`, `Depth`, `Weight`, `BBranch`, are used to control the number of conditional branches and the execution probability of the buggy branch. The other two parameters, `Iteration` and `Has_Data_Constraint`, are used to control the generation of programs with bugs reside in a deep loop or recursive call.

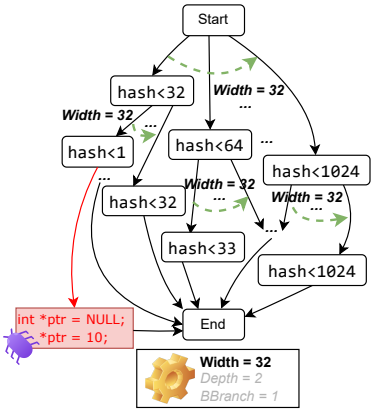
Number of conditional branches. As discussed in Section 3.1, we abstract each program as a control flow graph. Figure 1 (see Section 1) and Figure 3 demonstrate the program skeletons (Figures 1(a) and 3(a)) and control-flow graphs (Figures 1(b) and 3(c)) of the programs generated for the `Depth` and `Width` parameters. The parameters `Width` and `Depth` are used to quantify the horizontal and vertical complexities of the graph, with

```
1  void COMP_W$Width_D2_B1(unsigned hash)
2  {if (hash < $Width*1) {// level1
3      if (hash < 1) {// level2
4          PRINTF("This is branch 1\n");
5          /* Insert a bug here */
6      } else if (hash < 2) {// level2
7          ...
8      } else if (hash < $Width) {// level2
9          PRINTF("This is branch $Width\n");
10     }
11 } else if (hash < $Width*2) {// level1
12         ...
13 } else if (hash < $Width*$Width) {// level1
14     if (hash < $Width*($Width-1)+1) {// level2
15         PRINTF("This is branch
16                 $Width*($Width-1)+1\n");
17         ...
18     } else if {hash < $Width*($Width-1)+$Width} {
19         ... // level2
20     }
21 }
```

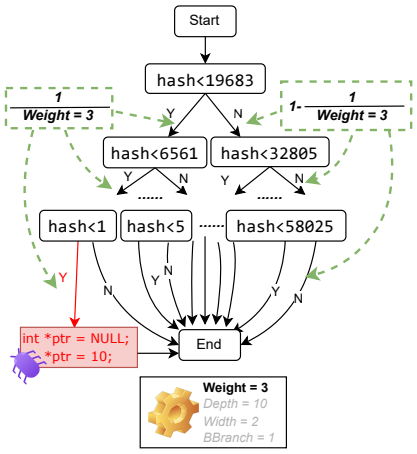
(a) Program skeleton (COMW).

```
1  void COMP_W2_D10_ω$Weight_B1(unsigned hash)
2  {if (hash < pow($Weight, 9})) {// level1
3      if (hash < pow($Weight, 8)) {// level2
4          ...
5              if (hash < 1) {// level10
6                  PRINTF("This is branch 1\n");
7                  /* Insert a bug here */
8                  ...
9  } else {// level1
10     if (hash < pow($Weight, 10)-pow($Weight, 8)*
11         pow($Weight-1, 2)) {// level2
12         ...
13     else {// level2
14         if (hash < pow($Weight, 10)-pow($Weight, 7)
15             *pow($Weight-1, 3)) {// level3
16             ...
17             if (hash < pow($Weight, 10)-
18             pow($Weight, 0)*
19             pow($Weight-1, 10)) {// level10
20                 ...
21 }
```

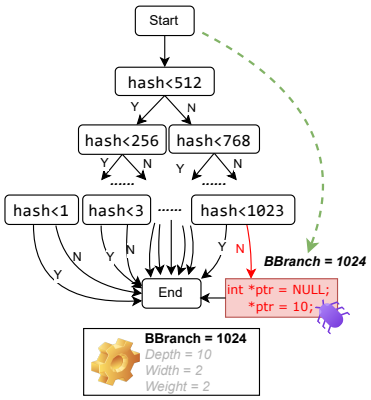
(b) Program skeleton (COMWE).



(c) Control-flow graph (COMW).



(d) Control-flow graph (COMWE).



(e) Control-flow graph (COMB).

Fig. 3. Program skeletons and control-flow graphs for control-flow complexity features (I).

`Width` controlling the number of branches from each `if` condition and `Depth` representing the nesting level. Each `if` condition has the same number of outgoing branches and makes the control-flow graph a balanced tree. This structure ensures that each branch yields equal code coverage, so that the probability of executing any given conditional branch is controlled by the `Weight` parameter, which will be discussed later.

In Figure 1(a), parameters `Depth`, `Width`, `Weight` and `BBranch` are denoted as D, W, ω and B, respectively. The programs generated for the `Depth` parameter have a variable `Depth` that can be configured to any positive integer, while other parameters are set to default values (`Width` (2), `Weight` (2) and `BBranch` (1)). When `Depth` grows, the nesting level of `if` conditions increases as illustrated in Figure 1(a) (e.g., lines 2-8 and lines 12-18), which results in a deeper control-flow graph. The green curly brace in Figure 1(b) illustrates the depth of control-flow graph. The constraints for each `if` condition are calculated as $\texttt{hash} < 2^{\texttt{Depth}-\texttt{Level}}$ or $\texttt{hash} < 2^{\texttt{Depth}} - 2^{\texttt{Depth}-\texttt{Level}}$ as shown in the program skeleton, where `Level` is the nesting level of the `if` condition. We generated 8 programs for the `Depth` parameter by varying it from 2 to 16 in increments of 2. We grouped these programs under the *COMD* folder in `FeatureBench` [1]. These parameter settings were decided empirically. For example, we chose 16 as the maximum setting for the `Depth` parameter to avoid generating too complex programs that may lead to excessive total possible paths ($2^{16} = 65,536$) and long compilation time (more than 1 hour). We chose 2 as the fix value for `Width` to minimize the impact of this parameter on the fuzzing performance. Figure 1(b) shows the control-flow graph of a program from the *COMD* group, with `Depth` set to 10, yielding $2^{10} = 1024$ possible paths.

The programs generated for the `Width` parameter have a variable `Width` that can be configured to any positive integer, while other parameters are set to default values (`Depth` (2) and `BBranch` (1)), shown in Figure 3(a). Note that for programs with a `Width` greater than 2, the `Weight` parameter is not considered, and we ensure that each conditional branch has an equal probability of execution. As the `Width` parameter grows, the number of branches from each `if` condition increases, as shown in Figure 3(a) (e.g., lines 3-8), which results in a wider control-flow graph. The green arrows in Figure 3(c) illustrate the width of the control-flow graph. For the first nesting level, the constraints for each `if` condition are calculated as `hash` < `Width` × `N`, where `N` represents the `N`th conditional branch at current nesting level. For the second nesting level, the constraints are calculated as `hash` < `Width` × `(N-1)` + `M`, where `N-1` represents the `N-1`th conditional branch at first nesting level and `M` represents the `M`th conditional branch at the second nesting level. We generated 16 programs for the `Width` parameter by varying it from 32 to 256 in increments of 16, and grouped them under the *COMW* folder in `FeatureBench` [1]. The increment value for `Width` is larger than that for `Depth` because `Depth` grows the total possible paths in an exponential manner. Therefore, it has a more significant impact on the program size than `Width`. The total number of possible paths of the largest program for the `Width` parameter is also 65,536 (256^2). Figure 3(c) shows the control-flow graph of a program from the *COMW* group, with `Width` set to 32, yielding $32^2 = 1024$ possible paths.

The larger the `Width` and `Depth`, the more possible paths the fuzzer may need to traverse to reach the buggy code. These programs use the `hash` variable to determine the execution flow. This variable is calculated by summing the hash values of each character in a fuzzing mutant, dividing by $\texttt{Width}^{\texttt{Depth}}$, and taking the remainder, which determines the executed branch. Each program contains a bug caused by a null pointer dereference, leading to a crash when executed. The location of the injected bug is controlled by the `BBranch` parameter, which determines the branch where the bug will be injected. Figure 1(a) (line 7) and Figure 3(a) (line 5) show the placeholders for bug injection. For programs generated with the `Depth` and `Width` parameters, the bug is always injected into the first branch to minimize the impact of bug location on the probability of triggering the bug. The control-flow graphs in Figure 1(b) and Figure 3(c) illustrate the specific bug being injected into the first branch, which corresponds to the branch where the leftmost leaf node is located. In `FeatureBench`, we generated 24 (16 for *COMW* and 8 for *COMD*) programs for this feature (see Table 1).

Execution probability of conditional branches. Figures 3(b) and 3(d) demonstrate the program skeletons and control-flow graph of the programs generate for the `Weight` parameter. The `Weight` parameter controls the probability of each conditional branch being executed. which is denoted as ω in Figure 3(b). Note that `Weight` parameter does not equate to the probability of the branch being executed. The probability can be calculated as $\frac{1}{\texttt{Weight}^{\texttt{Depth}}}$. The expectation is that the lower the probability is, for the fuzzers without any smart seed selection strategy, the less likely the branch will be executed to trigger the bug.

In Figure 3(b), the generated programs have a variable `Weight` that can be configured to any positive integer, while other parameters `Width`, `Depth`, and `BBranch` are set to 2, 10, and 1, respectively. This ensures that the size of the base program is neither too small, which would allow fuzzers to reach the bug too easily, nor too large, which would lead to long compilation times and could result in excessively large numbers being compared to `hash` in the `if` conditions, potentially increasing the data-flow complexity, which will be discussed in Section 4.2. Setting `BBranch` to 1 ensures the bug is always injected in the first branch. We manipulate the number that `hash` is compared to in each `if` condition to control the execution probability of each branch. For each nesting level, the constraints for `if` conditions are calculated as $\texttt{hash} < \texttt{Weight}^{\texttt{Depth-Level}}$ or $\texttt{hash} < \texttt{Weight}^{\texttt{Depth}} - \texttt{Weight}^{\texttt{Depth-Level}} \times \texttt{Weight-1}^{\texttt{Level}}$. We generated 7 programs for the `Weight` parameter by varying it from 2 to 8 in increments of 1. These programs are grouped under the *COMWE* folder in `FeatureBench` [1]. Figure 1(d) shows the control-flow graph of a program from the *COMWE* group, with `Weight` set to 3. The probability of every `True` branch (denoted as `Y`) being executed is $\frac{1}{\texttt{Weight}} = \frac{1}{3}$, while the probability of every `False` branch (denoted as `N`) is $1 - \frac{1}{\texttt{Weight}} = \frac{2}{3}$. The probability of the buggy branch being traversed is $\frac{1}{\texttt{Weight}^{\texttt{Depth}}}$, which is $\frac{1}{3^{10}}$ in this case.

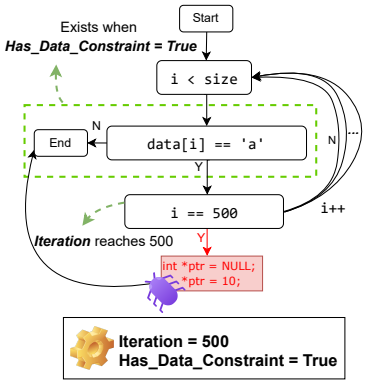
The `BBranch` parameter determines on which branch the bug is located, which can serve as another factor influencing the probability of reaching the buggy code. The skeleton of programs with the varying `BBranch` parameter is same as the one shown in Figure 1(a), except that `BBranch` parameter can be configured to any positive integer not larger than $\texttt{Weight}^{\texttt{Depth}}$, and `Width`, `Depth`, and `Weight` are set to default values 2, 10, and 2, respectively. We generated 32 programs for the `BBranch` parameter by varying it from 1 to 1024 in increments of 32, grouped under the *COMB* folder in `FeatureBench` [1]. Figure 3(e) shows the control-flow

```
1  void LOOP_I$Iteration
2  (unsigned char *data,
3               long size) {
4  for (unsigned int i = 0;
5               i < size; i++) {
6      if (data[i] == $MAGIC) {
7          if (i == $Iteration) {
8        /* Insert a bug here */
9          }
10     } else { break; } }}
```

(a) Program skeleton (LOOP).

```
1  void RECUR_I$Iteration
2  (unsigned char *data,
3               long size, int i)
4  { if (data[i] == $MAGIC) {
5      if (i == $Iteration) {
6      /* Insert a bug here */
7      }
8      RECUR_I$Iteration
9      (data, size, i + 1); } }
```

(b) Program skeleton (RECUR).



(c) Control-flow graph (LOOPDI).

Fig. 4. Program skeletons and control-flow graphs for control-flow complexity features (II).

graph of a program from the *COMB* group, with parameter `BBranch` set to 1024. We generated 39 (7 for *COMWE* and 32 for *COMB*) programs for this feature (see Table 1).

Loops and recursions. The parameter `Iteration` controls the number of the iterations of loops and recursions. Each program has a bug injected, guarded by the loop or recursive call, which can only be triggered when the iteration count reaches a specific value. Additionally, we incorporate data-flow complexity into this feature by adding data constraint checks before reaching the buggy code, to increase the difficulty of these test cases. The binary parameter `Has_Data_Constraint` controls whether the data constraints are incorporated into the program. Figures 4(a) and 4(b) illustrate two program skeletons that are controlled by these two parameters. Figure 4(a) demonstrates the skeleton when the bug is injected within a loop. The variable `data` is the fuzzing input and the `size` represents the length of the input. The program checks if the current loop iteration `i` equates to the `Iteration` parameter (line 7), and if so, the bug is triggered (line 8). The presence of the `if` check highlighted in green (line 6) is controlled by the boolean parameter `Has_Data_Constraint`. When `Has_Data_Constraint` is enabled, the data constraint check must also be satisfied to trigger the bug. The `MAGIC` is a randomly generated character that must appear in the fuzzing input to meet this constraint. Specifically, when `Has_Data_Constraint` is on, the input must contain a sequence of consecutive `Iteration` characters of `MAGIC` to pass the data constraint check. Figure 4(b) shows the recursive version of the same program, which is controlled by the same parameters.

For both types *loop* and *recursion*, we started with 5 and 10, and multiply by orders of 10, up to 50,000 and 100,000, as the settings for the `Iteration` parameter, resulting in 20 (10+10) programs. These programs are grouped under the *LOOPI* and *RECURI* folders in `FeatureBench` [1]. We chose values in the multiplication of order of 10 to create a significant gap in memory consumption between programs, allowing us to better distinguish the performance of fuzzers. The upper bound for `Iteration` is set to 100,000, as most fuzzers have reached their limits by this point, enabling us to draw meaningful conclusions about their performance. For the data-constrained variants discussed above, we set the `Iteration` values from 50 to 500 in increments of 50, resulting in another 20 (10+10) programs. These programs are grouped under the *LOOPDI* and *RECURDI* folders in `FeatureBench` [1]. We chose significantly lower upper bounds for these programs because the data constraint checks significantly increase the difficulty of triggering the bug. Figure 4(c) shows the control-flow graph of a program from the *LOOPDI* group, with `Iteration` set to 500 and `Has_Data_Constraint` enabled. The program has a bug injected in the loop, which can only be triggered when the loop iteration count reaches 500 and the input contains 500 consecutive magic characters. We generated 40 (10 for *LOOPD*, 10 for *RECURD*, and 10 each for their data constraint counterparts, *LOOPDI* and *RECURDI*, respectively) programs for this feature.

4.2 Data-Flow Complexity

In Section 3, we extracted three program features that can represent the data-flow complexity of the target program: *magic bytes*, *checksum tests*, and *nested magic bytes or checksum tests*. To define the data-flow complexity of the programs, we use four parameters: `Start`, `Length`, `Depth`, and `Count`. We generate programs where a bug is guarded by one or more complex data-flow conditions along the execution paths. Figure 5 shows four program skeletons and four control-flow graphs that are controlled by these four configurable parameters.

```
1 void MAGIC_S$Start_L$Length_D1
2 (unsigned char *data, long size)
3 {if (strncmp((char *)(data + $Start),
4       $MAGIC_BYTES , $Length) == 0) {
5     ... /* Insert a bug here */ }
```

(a) Program skeleton (MAGICS, MAGICL).

```
1 void CHECKSUM_C$Count_D1
2 (unsigned char *data, long size)
3 {if ($CHECKSUM_TEST1 &&
4       $CHECKSUM_TEST2 &&
5       $CHECKSUM_TEST3) {// C=3
6     ... /* Insert a bug here */ }
```

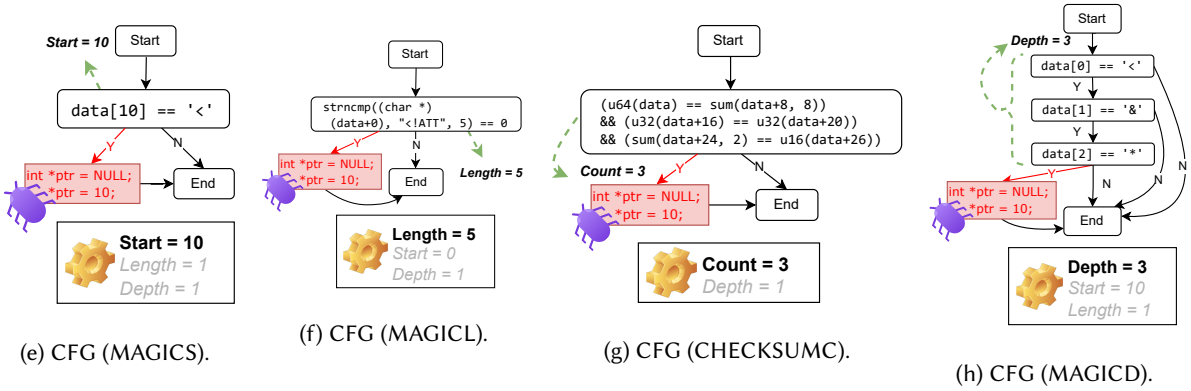
(b) Program skeleton (CHECKSUMC).

```
1 void MAGIC_S0_L1_D$Depth
2 (unsigned char *data, long size)
3 {if ((data[0]) == $MAGIC1) {
4     if ((data[1]) == $MAGIC2) {
5         if ((data[$Depth-1]) == $MAGIC3) {// D=3
6             ... /* Insert a bug here */ }
```

(c) Program skeleton (MAGICD).

```
1 void CHECKSUM_C1_D$Depth
2 (unsigned char *data, long size)
3 {if ($CHECKSUM_TEST1) {
4     if ($CHECKSUM_TEST2) {
5         if ($CHECKSUM_TEST3) {// D=3
6             ... /* Insert a bug here */ }
```

(d) Program skeleton (CHECKSUMD).



(e) CFG (MAGICS). (f) CFG (MAGICL). (g) CFG (CHECKSUMC). (h) CFG (MAGICD).

Fig. 5. Program skeletons and control-flow graphs for data-flow complexity features.

Magic bytes. The `Start` and `Length` parameters are used for generating the *magic bytes* type of complex data-flow conditions. The *magic bytes* condition checks if a sequence of characters in the input matches the magic string/character defined in the condition. `Start` defines the starting index of the magic bytes in the input, while `Length` defines the number of magic characters involved to satisfy the condition. Figure 5(a) shows the skeleton where bug is guarded by a magic bytes condition. The variable `data` is the fuzzing input and the `size` represents the length of the input. The program checks if the fuzzing input contains a magic string/character that starts at index `Start` and has a length of `Length` (lines 3-4). The `Depth` parameter is set to 1 in this case, meaning that only one level of condition needs to be satisfied. The `MAGIC_BYTES` is a randomly generated string of length `Length` that must appear in the fuzzing input in order to meet this condition.

To create programs that test the impact of the starting index, we set the `Length` parameter to 1 to avoid the impact of String length on the fuzzing performance, and vary the `Start` parameter from 0 to 90 in increments of 10, which results in 10 programs that are grouped under the *MAGICS* folder in `FeatureBench` [1]. We also set the `Start` to 0 and vary the `Length` from 1 to 10 in increments of 1, which results in another 10 programs that are grouped under the *MAGICL* folder in `FeatureBench` [1]. We chose these settings because most fuzzers seem to have reached their limits of handling lengthy magic bytes and inputs when the length of magic characters reaches 10 and the position of the magic bytes exceeds 90th character within the input. Figures 5(e) and 5(f) each shows a control-flow graph of a program from the *MAGICS* and *MAGICL* groups, respectively. Figures 5(e) shows a program with `Start` set to 10 and `Length` set to default value 1. `data[10]` is compared to the magic character `<` to guard the buggy code. Figure 5(f) shows a program with `Length` set to 5 and `Start` set to default value 0. `strncmp((char *)(data+0), "<!ATT", 5) == 0` compares the first 5 characters of the input to the magic string `<!ATT` to guard the buggy code. We generated 20 (10 for *MAGICS* and 10 for *MAGICL*) programs for the *magic bytes* feature, shown in Table 1.

Checksum tests. The `Count` parameter is used for generating the *checksum tests* type of complex data-flow conditions. It defines the number of checksum tests that guard the buggy code. The *checksum tests* validates if certain characters of the input satify the predefined checksum tests. In real-world scenarios, such tests are often

used to detect data corruption (e.g., in network programs). We use the logical-AND (`&&`) operator to combine multiple tests in a single `if` check. Figure 5(b) shows the program skeleton where the bug is guarded by three checksum tests (lines 3-5) and Figure 5(g) shows the corresponding control-flow graph. The `Count` parameter is set to 3 in this case, meaning that three conditions are combined with the `&&` operator, and all of them need to be satisfied to reach the buggy code (line 6). The `Depth` is fixed to 1, meaning that only one level of condition needs to be satisfied. Each `CHECKSUM_TEST` is a manually crafted checksum test that defines a specific data constraint. For example, `(average(data, 4) == sum(data+4, 8))` checks whether the average of the first 4 bytes of the input data is equal to the sum of the next 4 bytes. The character ranges in the fuzzing input checked by each `CHECKSUM_TEST` do not overlap, ensuring that no conflicting constraints will occur. In total, we generated 10 checksum tests to use as the `CHECKSUM_TEST` in program templates with `Count` ranging from 1 to 10 in increments of 1. We stopped at 10 because a condition check with more than 10 checksum tests is rarely seen in real-world programs, and most fuzzers seem to have reached their limits of handling complex checksum tests when number of tests reaches 10. Each checksum test applies one of several operations, such as sum, average and product on 2, 4, or 8 bytes of fuzzing input, creating constraints through combinations of these operations. These programs are grouped under *CHECKSUMC* folder in `FeatureBench` [1], and also reflected in Table 1.

Nested magic bytes and checksum tests. The `Depth` parameter is used for generating the *nested magic bytes and checksum tests* type of complex data-flow conditions. It defines the nesting level of conditions that must be met before reaching the buggy logic. Figures 5(c) and 5(d) show the program skeletons of *nested magic bytes* and *nested checksum tests*, respectively. For demonstration purpose, we set the `Depth` to 3 in both cases, meaning that three nested conditions are defined to guard the buggy code. The `MAGIC1`, `MAGIC2`, and `MAGIC3` in Figure 5(c) are randomly generated characters that must appear in the fuzzing input at the index of 0, 1, ..., and `Depth-1`, respectively (lines 3-5). The `Start` and `Length` parameters are fixed to 0 and 1, respectively, meaning that the magic bytes are expected to start from the beginning of the input and only one character is checked for each magic byte condition. This minimizes the impact of these parameters on the fuzzing performance, allowing us to focus on the effect of only the nesting depth. Similarly, the `Count` parameter is fixed to 1 in Figure 5(d), meaning that only one checksum test is checked at each nesting level, to avoid the impact of the number of checksum tests. We vary the `Depth` from 1 to 10 in increments of 1 for both types, which results in 10 programs each. We group these programs under the *MAGICD* and *CHECKSUMD* folders in `FeatureBench` [1]. We believe 10 is a reasonable upper bound for the `Depth` parameter, as most fuzzers seem to be struggling to reach the buggy code when the nesting depth reaches 10. Figure 5(h) shows the control-flow graph of a program from the *MAGICD* group, with `Depth` set to 3. `data[0]`, `data[1]`, and `data[2]` are compared to the magic characters `<`, `&`, and `*`, respectively, to guard the buggy code. In total, we generated 20 (10 for *MAGICD* and 10 for *CHECKSUMD*) programs for the feature of *nested magic bytes and checksum tests*, shown in Table 1.

5 Evaluation

5.1 Experimental Setup

Fuzzer selection. We choose 11 fuzzers as they represent the improvements from which the program features implemented in `FeatureBench` are extracted and they are popular coverage-guided grey-box fuzzers used in the community. EcoFuzz [53] and MOpt [31] represent fuzzers that improve efficiency through smart strategies. AFLFast [4] and FairFuzz [25] are designed to prioritize seeds that cover infrequent code regions. TortoiseFuzz [50] and Memlock [51] are memory information-guided fuzzers that are expected to be sensitive to vulnerable control-flow structures such as loops and recursions. RedQueen [3] and Laf-intel [22] are designed to handle complicated hard checks such as magic bytes and checksum tests. Finally, we include popular coverage-guided fuzzers from FuzzBench [35], such as AFL [54], AFL++ [10], and Honggfuzz [13]. These fuzzers are frequently used as baselines in fuzzing evaluations [27, 34, 53, 55, 59]. For two fuzzers, we evaluated multiple variants of their configurations. Specifically, for TortoiseFuzz, we experiment with two of its coverage metrics, `bb` (Tort-B) and `loop` (Tort-L). The `bb` metric counts the security-sensitive edges at the basic block granularity, and the `loop` metric counts the security-sensitive edges based on if it is a back edge. For Memlock, we run both of its variants, `stack` (Mem-S) and `heap` (Mem-H), which utilize the stack memory usage and heap memory usage to guide the fuzzing process, respectively. These fuzzers and their associated features are summarized as follows. The abbreviated names of the fuzzers, if any, are shown within the parentheses.

- Number of conditional branches: EcoFuzz (Eco) [53], MOpt [31].
- Execution probability of conditional branches: AFLFast [4], Fairfuzz (Fair) [25].
- Loops and recursions (with data constraints): TortoiseFuzz (Tort) [50], Memlock (Mem) [51].

- (Nested) magic bytes and checksum tests: RedQueen (Red) [3], Laf-intel (Laf) [22].
- Other popular coverage-guided fuzzers from FuzzBench [35]: AFL [54], AFL++ [10], Honggfuzz (Hongg) [13].

Metrics. In our experiments, we set fuzzers to stop fuzzing once the injected bug is found, and collect the running time of each fuzzer to trigger the crash. We ran each fuzzer on each benchmark program with a 2-hour timeout and for 20 repeated trials. The 2-hour timeout is sufficient because the programs generated in `FeatureBench` are relatively simple, and most fuzzing trials can complete, i.e., find the bug, within seconds or minutes. Those that time out after 2 hours would clearly indicate that the fuzzer does not handle the corresponding feature effectively. Therefore, to show how effectively each fuzzer supports specific feature parameters, we report the *completion rate* of each fuzzer for each feature parameter. The completion rate is calculated as the ratio of successfully completed programs (that do not time out) w.r.t. the total number of programs under the feature parameter. A completion rate of 1.0 indicates that the fuzzer was able to find the bug in all the programs with that specific feature parameter.

We also calculate the *Spearman's rank correlation coefficient* [48] of each feature parameter (except for the *BBranch* parameter) and the fuzzing runtime to analyze the impact of the strength of each parameter on the performance of different fuzzers. Spearman's correlation is a nonparametric measure of the strength and direction of association between two ranked variables. Spearman's correlation coefficient, denoted by ρ or r_s, assesses how well the relationship between two variables can be described using a monotonic function. The formula for Spearman's rank correlation coefficient is: $r_s = 1 - \frac{6 \sum d_i^2}{n(n^2-1)}$ where d_i is the difference between ranks for each pair of observations, and n is the total number of observations [48]. The value of r_s ranges from -1 to 1, where 1 indicates a perfect positive correlation, -1 indicates a perfect negative correlation, and 0 indicates no correlation. When we analyze the results, we consider those with correlation above the 0.7 threshold as having a strong correlation while those below 0.3 as weak correlations. This correlation analysis is appropriate to use because for all feature parameters, except for the *BBranch* parameter, the increase of their absolute values means an increase of the strength of the corresponding features. Therefore, a stronger positive correlation means that the fuzzer's performance gets worse as the strength of the feature parameter increases (e.g., a fuzzer takes longer time to find a bug when this bug is injected in a deeper recursion). And a stronger negative correlation indicates that the fuzzer's performance gets worse as the strength of the feature parameter decreases (e.g., a fuzzer takes longer time to find a bug when this bug is injected in a branch with lower weight). We implemented Spearman's correlation coefficient using the `spearmanr` function from Python's `scipy.stats` module [45]. The results are presented along with a *p-value*, which indicates the statistical significance of the correlation. A *p-value* less than 0.05 indicates a statistically significant correlation.

For the *BBranch* parameter, we perform the *Mann-Whitney U Test* [33] for each fuzzer to analyze the statistical significance of the differences in the runtime of fuzzers on programs with different bug locations. The *Mann-Whitney U test* (also known as the *Wilcoxon rank-sum test*) is a nonparametric test used to assess whether there is a significant difference between the distributions of two independent samples. We calculate the *p-value* for each program pair to determine if the differences in runtime are statistically significant (less than 0.05). The results are visualized in a heatmap.

Research questions. We answer two research questions in this evaluation:

- **RQ1:** How well do the fuzzers perform on each program feature in `FeatureBench`?
- **RQ2:** With the assistance of data visualization, can we confirm expected and identify unexpected or previously unknown fuzzing behavior associated with program features?

To answer RQ1, we perform the correlation analysis or Mann-Whitney U test to understand the impact of each program feature parameter on the performance of different fuzzers. We also report the completion rate of each fuzzer for each feature parameter to show how effectively each fuzzer supports the feature parameters. To answer RQ2, we report and visualize the median result of each fuzzer over the 20 trials for each program feature and generate observations that may or may not align with the common wisdom in the fuzzing literature to further demonstrate the usefulness of a feature-based benchmark like `FeatureBench`.

Hardware environment. All experiments were conducted on a server with an AMD Ryzen Threadripper PRO 5975WX CPU (64 threads) and 128GB RAM, running Ubuntu 22.04.

5.2 RQ1: How Well Do the Fuzzers Perform on Each Program Feature?

Tables 2 and 3 show each fuzzer's correlation and completion rate for the control-flow and data-flow features. The **corr** columns show the Spearman correlation coefficient, while the **comp** column show the completion rate.

Table 2. Spearman correlation and completion rate for control-flow complexity features.

Fuzzer	COMD		COMW		COMWE		COMB		LOOPI		LOOPDI		RECURI		RECURDI	
	corr	comp	corr	comp	corr	comp	corr	comp	corr	comp	corr	comp	corr	comp	corr	comp
Eco	**0.287***	1.00	**-0.024**	1.00	**-0.237***	1.00	-	1.00	0.895*	0.90	0.712*	1.00	0.857*	1.00	0.653*	1.00
MOpt	0.662*	1.00	**0.106**	1.00	**-0.192***	1.00	-	1.00	0.926*	1.00	0.663*	1.00	0.895*	1.00	0.630*	1.00
AFLFast	0.517*	1.00	**0.010**	1.00	-0.307*	1.00	-	1.00	0.027	0.50	0.768*	0.80	0.358*	0.50	0.806*	0.20
Fair	0.513*	1.00	**0.141***	1.00	-0.364*	1.00	-	1.00	0.896*	0.80	0.762*	1.00	0.895*	0.80	0.705*	1.00
Red	0.878*	1.00	-	0.06	-0.452*	1.00	-	1.00	**0.074**	1.00	**0.054**	1.00	**0.151***	1.00	**0.023**	1.00
Laf	0.853*	0.75	0.333*	0.38	**-0.291***	1.00	-	1.00	**0.106**	1.00	**0.068**	1.00	**0.184***	1.00	**0.038**	1.00
Mem-S	0.534*	1.00	**0.154***	1.00	**-0.228***	1.00	-	1.00	0.875*	1.00	0.533*	1.00	0.827*	0.90	0.522*	1.00
Mem-H	0.500*	1.00	**0.177***	1.00	**0.061**	1.00	-	1.00	0.879*	1.00	0.541*	1.00	0.880*	1.00	0.428*	1.00
Tort-B	0.839*	1.00	0.253*	0.94	-0.474*	1.00	-	1.00	0.918*	1.00	0.525*	1.00	0.898*	0.90	0.552*	1.00
Tort-L	0.735*	0.88	0.044	0.50	-0.410*	1.00	-	1.00	-0.109	0.20	0.494*	1.00	-0.116	0.20	0.525*	1.00
AFL	0.640*	1.00	**0.255***	1.00	-0.315*	1.00	-	1.00	0.923*	1.00	0.754*	1.00	0.882*	1.00	0.681*	1.00
AFL++	0.894*	1.00	0.872*	1.00	-0.507*	1.00	-	1.00	-0.140	0.20	0.563*	1.00	0.000	0.20	0.562*	1.00
Hongg	0.366*	1.00	0.013	0.94	**-0.093**	1.00	-	1.00	0.325*	0.70	0.213*	0.70	0.212*	0.70	**0.187**	1.00

We determine if a correlation is statically significant based on the *p-value* as discussed in Section 5.1, and denote all statically significant correlations with an asterisk (*). Weak correlations (between -0.3 and 0.3) with a 100% completion rate are highlighted in bold, and a hyphen (-) indicates unavailable correlations due to insufficient data. For example, RedQueen only detected the bug in one program for COMW, making it impossible to calculate a meaningful correlation; such result is reflected in the (low) completion rate metric.

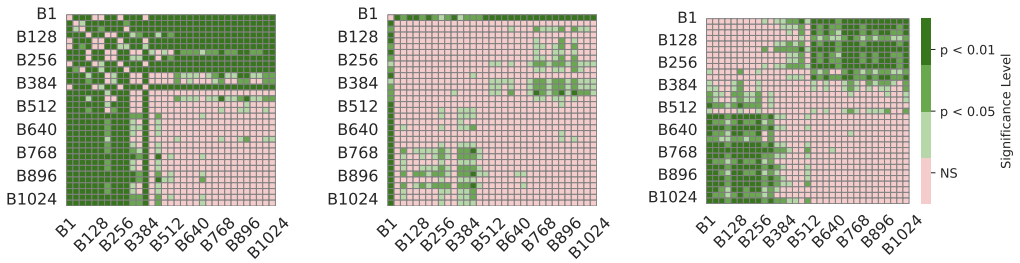
5.2.1 Control-Flow Complexity. In Table 2, COMD, COMW, COMWE, and COMB columns represent the results running on the programs generated by varying the `Depth`, `Width`, `Weight`, and `BBranch` parameters of control-flow complexity, respectively. LOOPI and RECURI represent programs generated by varying the `Iterations` parameter of loops and recursion, while LOOPDI and RECURDI are their counterparts that incorporate data-flow complexity as discussed in Section 4.1. Figure 6 presents three heatmaps, each corresponding to a different fuzzer (AFL++, Honggfuzz, and MOpt), to visualize the significance (*p-value*) of pairwise fuzzing time differences as the `BBranch` parameter varies.[1]

Depth of control-flow compexity (COMD). As shown in Table 2, the depth of control-flow complexity (COMD) has statistically significant correlations with the performance of all fuzzers. The correlation coefficients range from 0.287 to 0.894, with AFL++ showing the strongest positive correlation, and EcoFuzz showing the weakest positive correlation. RedQueen, Laf-intel, and the two variants of TortoiseFuzz also exhibit strong positive correlations. The strong positive correlation indicates that as the depth of the control-flow complexity increases, AFL++, RedQueen, Laf-intel, and TortoiseFuzz will be affected the most, resulting in longer runtimes, while EcoFuzz is the least affected by this parameter of control-flow complexity. The completion rates for all fuzzers are very high. Most fuzzers have a 100% completion rate, indicating that they can successfully find bugs in all programs for this feature parameter. Laf-intel has the lowest completion rate of 0.75, which timed out when D equals to 14 and 16, indicating that it does not scale as well as the other fuzzers when the depth of control-flow complexity grows to a high level.

Width of control-flow complexity (COMW). Varying the width of control-flow complexity (COMW) does not show statistically significant correlation with many fuzzers we evaluated. The significant correlations are not as strong as those for COMD, with correlation coefficients ranging from -0.024 to 0.872. Most fuzzers do not show a strong correlation with the width of control-flow complexity, indicating that the width of control-flow complexity does not significantly impact most fuzzers' performance, except for AFL++, demonstrating a strong positive correlation of 0.872. EcoFuzz, MOpt, AFLFast, Hongfuzz and TortoiseFuzz (loop) show no significant correlation with the width of control-flow complexity, indicating that they are not sensitive to the variation of this feature parameter. Interestingly, RedQueen does not have available correlation data for this feature parameter because it only successfully detected the bug in one program (W = 16). Laf-intel has a correlation of 0.333 with a low completion rate of 0.38. These two fuzzers are designed to excel at handling complicated hard checks as discussed in 3.2, but seem to struggle when the width of the conditional branches increases.

Weight of control-flow complexity (COMWE). Recall that as the `Weight` parameter increases, the probability to reach the buggy branch decreases (Section 4.1). Interestingly, in Table 2, we observe that fuzzers take shorter

[1]The heatmaps of all fuzzers and all other experimental data are available in our artifact [1].



(a) AFL++ (b) Honggfuzz (c) MOpt

Fig. 6. Mann-Whitney U test heatmap (*p-value*) for program pairwise runtime comparisons.

time to locate bugs as the probability of the buggy branches decreases; in other words, the fuzzers are slower to find the bug when the probability to reach the buggy branches is higher. The correlations between `Weight` and fuzzing performance are mostly negative, except for Memlock (heap), which shows a very weak positive correlation of 0.061. The negative correlation coefficients range from -0.507 to -0.093, with AFL++ showing the strongest negative correlation, and Honggfuzz showing the weakest negative correlation. This indicates AFL++ is the most sensitive to the increases in the probability of buggy branch. In contrast, Honggfuzz and Memlock (heap) show very low correlations with this parameter, suggesting that changes in the probability of buggy branches do not significantly affect their performance. The completion rates for all fuzzers are 100%.

BBranch of control-flow complexity (COMB). In Figure 6, we use heatmaps to visualize the results of the Mann-Whitney U Test for three fuzzers (AFL++, Honggfuzz, and MOpt) to analyze the impact of bug location on fuzzing performance. Each heatmap shows the statistical significance in runtime differences for program pairs based on bug location. Pink cells denote no significant runtime difference between program pairs. We selected AFL++, Honggfuzz, and MOpt as they show distinct performance patterns based on bug location. For AFL++, when the bug locates in a branch earlier in the program (below about 400 out of all 1024 branches), the performance of AFL++ varied significantly (the green cells on the left or the upper part of Figure 6(a)); when the bug location is in a branch that appears later in the program, AFL++ found these bugs in similar runtime (all the pink cells on the bottom-right). MOpt result also shows that the fuzzer's performance is stable in programs with the later bug locations; however, MOpt's performance in programs with earlier bug locations is also similar (the pink cells on the top-left of Figure 6(c)), while the performance on the programs with the later bug locations and with earlier bug locations is significantly different. In contrast, most cells for Honggfuzz are pink (Figure 6(b)), suggesting that bug location minimally impacts its performance. These interesting findings spawned us to investigate the generated mutants of these fuzzers. For example, we found that the frequency of the generated mutants of AFL++ reaching each branch is significantly different and follows some pattern (e.g., during one period of fuzzing, the branches located around the median of the 1024 branches are visited frequently and in certain order). We believe it is worth checking the implementation of these fuzzers to further investigate this behavior; this result gives another example highlighting the usefulness of evaluating on benchmarks like `FeatureBench`.

Loops and recursions iterations (LOOPI and RECURI). As shown in Table 2, most fuzzers show strong positive correlations with the iterations of loops and recursion (LOOPI and RECURI), with correlation coefficients ranging from -0.140 to 0.926 for loop and from -0.116 to 0.898 for recursion. EcoFuzz, MOpt, FairFuzz, RedQueen, AFL and both variants of TortoiseFuzz show strong correlations with both loop and recursion iterations. Meanwhile, the completion rates for these fuzzers are also very high, with a minimum of 0.8 (FairFuzz). The results indicate that the growth of iterations in loop and recursion significantly impacts the performance of these fuzzers, leading to longer runtimes. However, these fuzzers are still able to find bugs in most of the programs where the bugs are located in deep loops or recursion. AFL++, Laf-intel and AFLFast show very low correlation with the iterations of loops or recursion, at the same time the completion rates are also quite low, indicating that these fuzzers do not effectively finding bugs that require high iterations of loops or recursion. The two variants of Memlock also show no significant correlation with the iterations of loops and recursion, however the completion rates are 100% for both cases, indicating that they are good at finding bugs in programs with high iterations of loops or recursion and are not affected by the variation of this parameter.

Loops and recursions iterations with data-flow complexity (LOOPDI and RECURDI). The results for this group of programs show very similar trends to those of LOOPI and RECURI with a few exceptions. AFL++

Table 3. Spearman correlation and completion rate for data-flow features.

Fuzzer	MAGICS		MAGICL		MAGICD		CHECKSUMC		CHECKSUMD	
	corr	comp	corr	comp	corr	comp	corr	comp	corr	comp
Eco	0.566*	0.900	0.921*	0.300	0.933*	0.500	0.701*	1.000	0.648*	1.000
MOpt	0.440*	0.900	0.932*	0.300	0.902*	0.500	0.489*	1.000	0.510*	1.000
AFLFast	0.648*	0.900	0.951*	0.300	0.929*	0.500	0.744*	1.000	0.759*	1.000
Fair	0.638*	0.900	0.946*	0.300	0.828*	0.600	0.771*	1.000	0.788*	1.000
Mem-S	**0.048**	1.000	0.887*	0.200	0.943*	1.000	**0.037**	1.000	**0.033**	1.000
Mem-H	**0.124**	0.900	-	0.100	0.837*	0.500	**-0.017**	1.000	**0.000**	1.000
Tort-B	0.660*	1.000	0.896*	0.200	0.921*	1.000	0.836*	1.000	0.828*	1.000
Tort-L	0.723*	1.000	0.891*	0.200	0.900*	1.000	0.814*	1.000	0.860*	1.000
Red	0.545*	0.900	**-0.070**	1.000	0.932*	1.000	0.650*	1.000	0.619*	1.000
Laf	0.608*	0.900	0.803*	1.000	0.935*	0.900	0.537*	1.000	0.565*	1.000
AFL	0.801*	0.900	0.950*	0.300	0.947*	0.500	0.863*	1.000	0.790*	1.000
AFL++	0.452*	0.900	0.845*	0.200	0.921*	1.000	0.491*	1.000	0.563*	1.000
Hongg	0.691*	1.000	-0.338*	1.000	0.936*	1.000	**0.223***	1.000	**0.171***	1.000

and Laf-intel are able to achieve a high completion rate (100%) for both features with a medium correlation with the iteration parameter. This could attribute to the fact that the experimental settings selected for iteration in this group of programs are not as high as those in the previous group, considering that the data-flow complexity is also taken into account. The results again suggest that the challenge for these two fuzzers lies mainly in handling programs with high iterations of loops and recursion.

Summary: `Depth` *of control-flow complexity has a stronger impact on the fuzzing performance than* `Width` *of control-flow complexity. AFL++ is most sensitive to the increase of control-flow complexity, while EcoFuzz is the least sensitive. RedQueen and Laf-intel perform well on programs with high* `Depth` *but struggle with high* `Width`*. Many fuzzers are sensitive to the increase of* `Iteration` *in loops and recursion. Memlock excels on the programs with high iterations of loops and recursion, while AFL++, Laf-intel and AFLFast struggle the most.*

5.2.2 Data-Flow Complexity. In Table 3, MAGICS, MAGICL and MAGICD denote the programs generated by varying the `Start`, `Length` and `Depth` parameters of the magic bytes check. CHECKSUMC and CHECKSUMD represent the programs generated by varying the `Count` and `Depth` parameters of the checksum tests.

Magic bytes (MAGICS, MAGICL, and MAGICD). As shown in Table 3, TortoiseFuzz (loop) and AFL++ exhibit the strongest positive correlation with the `Start` parameter of magic bytes, with correlation coefficients of 0.723 and 0.801, respectively. This suggests that the position of magic bytes has a significant impact on their performance. In contrast, the two variants of Memlock show the lowest correlation with this parameter, at 0.048 and 0.124, indicating minimal sensitivity to the position of magic bytes. Completion rates for all fuzzers are very high, with all fuzzers achieving a completion rate of 0.9 or higher. Memlock (stack), TortoiseFuzz (loop and bb), and Honggfuzz are 100% successful in finding bugs in all programs.

The parameter `Length` of magic bytes shows strong correlations with the performance of most fuzzers, with correlation coefficients ranging from -0.338 to 0.951. The completion rates for most fuzzers are very low, indicating that most fuzzers struggle with resolving magic bytes of large length. RedQueen and Honggfuzz perform exceptional well with a completion rate of 1.0 and low correlation coefficients, indicating that the increase of length of magic bytes does not significantly impact their performance. Laf-intel also achieves 100% completion rate, but with a high correlation of 0.803, indicating that long magic bytes will lead to longer running time for Laf-intel; however, it still is able to resolve the long magic bytes to trigger the crash.

The `Depth` parameter of magic bytes shows strong correlations with fuzzer performance, ranging from 0.828 to 0.947. Memlock (stack), TortoiseFuzz (loop and bb), RedQueen, AFL++, and Honggfuzz achieve a 100% completion rate, indicating that these fuzzers perform better on programs with deeply nested magic bytes than others. Overall, the depth of magic bytes significantly impacts the performance of all fuzzers.

Checksum tests (CHECKSUMC and CHECKSUMD). The `Count` and `Depth` parameters of checksum tests show similar correlation strengths with fuzzing performance, with coefficients ranging from -0.017 to 0.863 for `Count` and from 0 to 0.860 for `Depth`. All fuzzers achieve a 100% completion rate for both parameters. Memlock (stack and heap) and Honggfuzz show very low correlation coefficients for both, indicating that the number and nesting level of checksum tests do not significantly impact their performance.

Summary: The length and nesting level of magic bytes have larger impact on the performance of fuzzers than the position of magic bytes. Most fuzzers struggle with programs that contain long magic bytes, while RedQueen, Honggfuzz and Laf-intel handle them well. Memlock(stack), TortoiseFuzz and AFL++, although struggling with the long magic bytes, are able to find bugs guarded by deeply nested magic bytes. All fuzzers perform well on programs with a large number and deep nesting level of checksum tests. Memlock and Honggfuzz are least affected by the increase of both parameters of checksum tests.

5.3 RQ2: Analyzing Fuzzing Behavior Dependent on Program Features

In this section, we inspect the fuzzers' performance in each program feature to observe behaviors that are expected, unexpected or previously unknown based on the common wisdom in the literature and the technical descriptions of the fuzzers. For each feature parameter, we collected the median runtime of each fuzzer over the 20 trials, and created line plots to illustrate the performance trends of each fuzzer with respect to these parameters.

5.3.1 *Control-Flow Complexity.*

Number of conditional branches. Figures 7(a) and 7(b) show the median runtime of each fuzzer on programs with varying `Width` and `Depth`, respectively. The x-axes in the figures show the values of the respective parameters and the y-axes show the median time each fuzzer took to detect the injected bug. We observe that most fuzzers maintain a relatively stable runtime across different widths and depths. Laf-intel and RedQueen timed out on the programs with width greater than 16 and 96, respectively, suggesting that these fuzzers struggle with handling programs that contain a high number of branch conditions, where the control flow splits into multiple possible execution paths. However, RedQueen did not time out on any programs with increasing depth, and Laf-intel only timed out on the program with depth 12 or greater, indicating that these two fuzzers perform better when exploitation is more needed than exploration. Honggfuzz timed out at the width of 256 and was outperformed by all other fuzzers at the depth of 16 (note that the y-axis in Figure 7(b) was customized to accommodate the high runtime of Honggfuzz.). Its performance on both depth and width experiments suggests that it struggles when the control-flow complexity increases to a certain level, which indicating it may not be the best choice for programs with high control-flow complexity. However, on the programs with a low control-flow complexity, Honggfuzz achieved an outstanding performance, better than any other fuzzer. AFL++ maintains the best performance on programs with a larger width, while it performed worse than most fuzzers as the depth of the program increases. This suggests that AFL++ may be more effective at exploring the program paths than exploiting them.

We summarize the discussed observations across fuzzers in Table 4, along with their classification in the **Status** column as expected (`EP`), unexpected (`UE`), or previously unknown (`PU`). An observation is labeled as `EP` if it aligns with common wisdom in the literature and/or explicitly claimed in the corresponding paper(s), `UE` if it contradicts prior claims, and `PU` if it is a new finding from our experiments that is not explicitly stated in literatures. For brevity, we denote the control-flow feature, the number of conditional branches as `C1` in **FID** column.

Execution probability of conditional branches. Figure 7(c) shows the median runtime of each fuzzer on programs with varying weight of the buggy branch. The results show that the fuzzers tend to find the bug faster when the bug is located in the branch that is more infrequently executed. AFL++, Laf-intel, and RedQueen show a larger increase in runtime than other fuzzers when the weight of the buggy branch increases (smaller `Weight`), indicating that they are more sensitive to the execution probability of the branches, while Honggfuzz maintains an excellent performance across all weights, which align with the trends observed in Figure 7(d). We further compared the region coverage and the number of mutants generated to trigger the bug across different `Weight` parameters. For fuzzers with low variance in fuzzing time, we observed a significant decrease in the number of mutants generated and coverage as the buggy branch's weight decreases (larger `Weight`). This suggests that, despite the small variance in runtime, other performance metrics indicate improved fuzzing efficiency. Specifically, these fuzzers are able to locate the bug with fewer mutants and lower coverage when the bug is in an infrequently executed branch. This observation validates the claimed improvement of fuzzers like AFLFast and Fairfuzz, which prioritize seeds that cover infrequent paths and branches.

Figure 7(d) shows the median runtime of each fuzzer on programs where bugs locate at different branches. Honggfuzz maintains a very low increase rate of runtime across different branches, indicating that it is less sensitive to the bug's location. AFL++, Laf-intel, and RedQueen exhibit the most fluctuating runtimes, suggesting a higher sensitivity to the changes in bug location. These observations of fuzzer behaviors are summarized in Table 4. We denote the feature *execution probability of conditional branches* as `C2` in the **FID** column.

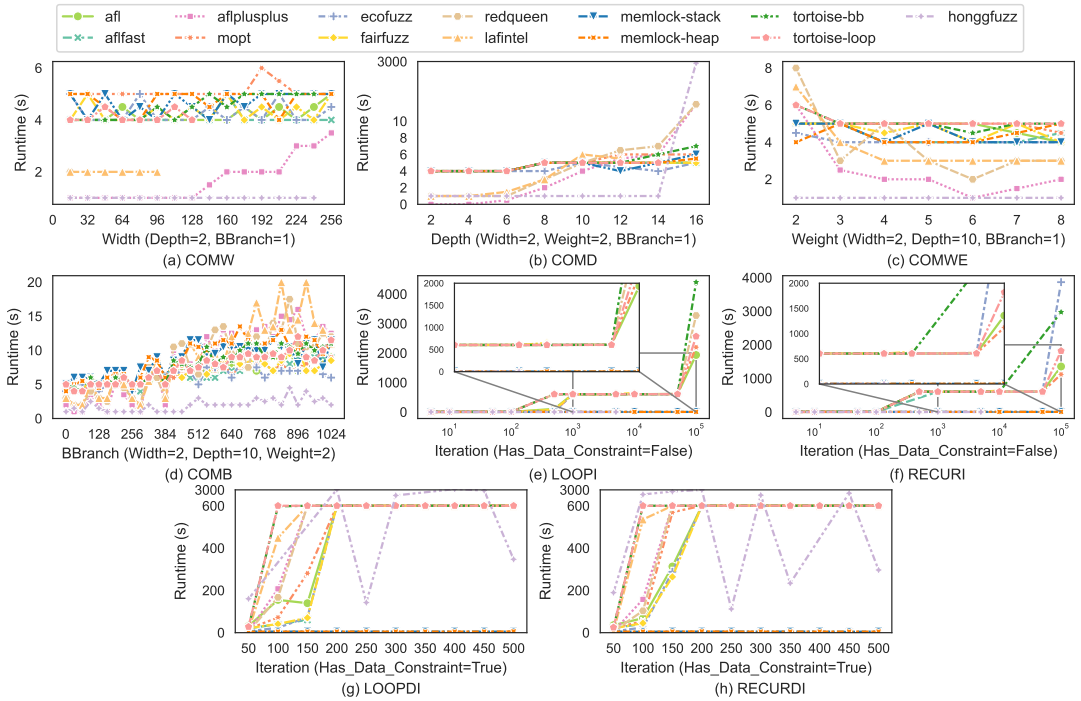


Fig. 7. Fuzzer runtime over `FeatureBench` programs with varying width (a), depth (b), buggy branch (c), weight (d), loop iteration (with data constraints) (e) and (g), recursion iteration (with data constraints) (f) and (h).

Loops and recursions. Figure 7(e) and Figure 7(f) show the median runtime of each fuzzer on programs with varying loop iterations and recursion iterations. The results show that most fuzzers exhibit similar performance trends and remain relatively stable for iterations up to 100. However, a significant increase in runtime is observed when iterations reach 500 or 1,000. Memlock (stack and heap) quickly finds bugs in all programs, even when loop and recursion iterations reach 100,000, while other fuzzers begin to struggle. Its performance aligns with the claim that memory-based guidance is effective in finding bugs in programs with vulnerable control-flow features like loops and recursion, which are often associated with high memory consumption. However, TortoiseFuzz (loop and bb) does not perform as effectively as expected. This may be attributed to its design, which prioritizes the presence of error-prone structures, such as loops, to guide its fuzzing process but does not account for the number of iterations. Consequently, TortoiseFuzz may miss opportunities to explore deeper loop iterations, limiting its effectiveness in finding bugs residing in high-iteration loops or recursions. Honggfuzz performs well with small number of loop or recursion iterations but experiences drastic performance drops, leading to timeouts as iterations increase to 10,000. AFL++, Laf-intel, and AFLFast do not effectively support high iterations as they time out at early stages of the experiments, at the iteration of 50 or 1000. Such performance aligns with the limitation that these traditional coverage-based grey-box fuzzers do not have awareness about memory-related information, thus struggle in finding bugs in programs with high loop and recursion iterations. We summarize the discussed observations across fuzzers in Table 4, and denote the feature *loops and recursions* as `C3` in the **FID** column.

Loops and recursions with data constraints. Figures 7(g) and 7(h) show the median runtime of each fuzzer on programs with varying loop and recursion iterations, respectively, incorporating data constraints. The y-axes in both figures are customized to accommodate high runtime of Honggfuzz, showing linear scale below 600 and logarithmic scale above 600. We observe that most fuzzers exhibit a significant runtime increase when the number of iterations reaches 100, 150, or 200, after which their perform remains stable. TortoiseFuzz (loop and bb) performed worse than AFL++, RedQueen and Laf-intel, and even further behind MOpt, Fairfuzz, AFL, AFLFast and EcoFuzz. Interestingly, Honggfuzz demonstrates unstable performance as iteration increases, causing large runtime fluctuations. This suggests that the introduction of data constraints has a greater impact on Honggfuzz than on other fuzzers. Memlock (stack and heap) consistently maintains the best performance across all programs with data constraints, indicating that memory-based guidance is highly effective when bugs are guarded by deeply nested loops or recursion, where memory consumption increases. In these cases, traditional coverage-guided fuzzers are

Table 4. Observations on control-flow complexity features. PU(Previously Unknown), EP(Expected), UE(Unexpected).

FID	Fuzzer	Observation	State
C1	AFL++	Perform better on high-width programs than high-depth programs.	PU
	Laf-intel, RedQueen	Perform better on high-depth programs than high-width programs.	PU
	Honggfuzz	Perform better on low-width/depth programs.	PU
C2	AFL++, Laf-intel, RedQueen	More sensitive to bug location changes and the weight of buggy branches.	PU
	Honggfuzz	Less sensitive to bug location changes and the weight of buggy branches and maintain better performance.	PU
	Fuzzers with low variance in fuzzing time (e.g., EcoFuzz, AFLFast)	Perform more efficiently on programs with lower buggy branch weight.	EP
C3, 4	Memlock	Perform excellently with higher loop/recursion iterations and data constraints.	EP
	TortoiseFuzz	Perform worse on programs with loops/recursions.	UE
	Honggfuzz	More sensitive to the introduction of data constraints.	PU
	Fuzzers other than Memlock	Perform worse on programs with higher loop/recursion iterations.	PU

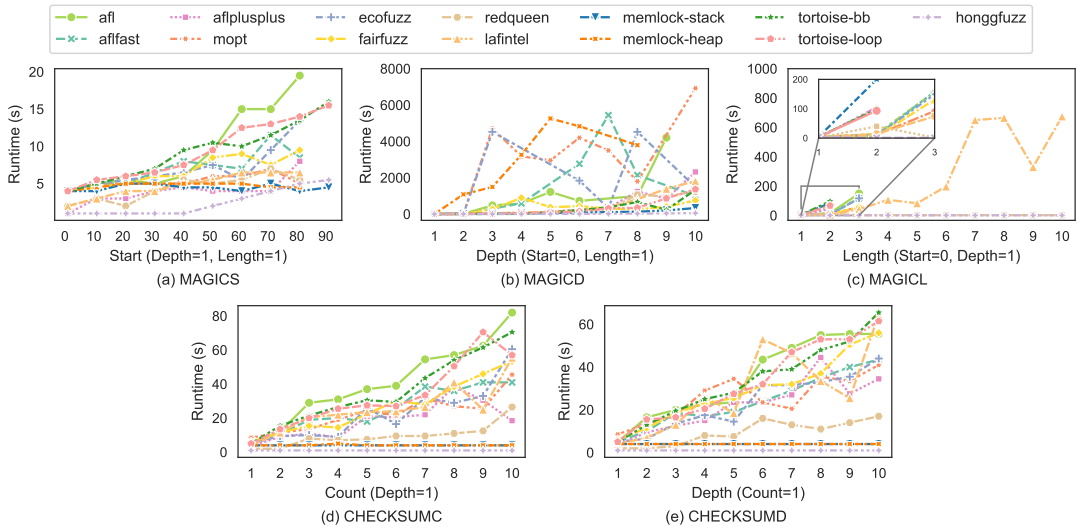


Fig. 8. Fuzzer runtime over `FeatureBench` programs with varying start index (a), depth (b), length (c) of magic bytes, count (d), and depth (e) of checksum tests.

less effective compared to Memlock. Both Memlock and TortoiseFuzz are designed to address control-flow features like loops and recursion. TortoiseFuzz leverages error-prone structures (e.g., loops) to guide its fuzzing process. However, its performance falls short in handling high iteration complexities, indicating limitations in managing challenges posed by large number of iterations, where Memlock excels. We summarize the discussed observations across fuzzers in Table 4, and denote the feature *loops and recursions with data constraints* as `C4` in **FID** column.

5.3.2 *Data-Flow Complexity.*

Magic bytes. Figure 8(a) shows median runtime of fuzzers on programs with varying start indexes of magic bytes with a single character. We observe that most fuzzers timed out on programs with start index of 80, indicating the upper bound limits of these fuzzers in handling long input strings. AFL, AFLFast, EcoFuzz, Fairfuzz do not perform as well as Laf-intel, Redqueen and AFL++ on programs with magic bytes locating at far index of inputs, showing that these fuzzers are less effective at handling long input strings. TortoiseFuzz, Memlock (stack) and Honggfuzz were able to make it to the most difficult test case with magic character locating at the 90th index of the input string. Memlock (stack) is the most effective fuzzer in this experiment and maintains the best performance.

Figure 8(b) shows median runtime of fuzzers on programs with varying levels of nesting magic byte checks. AFL, AFLFast, EcoFuzz and MOpt performed worse than Laf-intel, Redqueen and AFL++ on programs with nesting magic byte checks. EcoFuzz shows worse ranking than previous experiments, suggesting that EcoFuzz is less effective at finding bugs that locate in deep path guarded by nesting hard checks. However, the ranking of Fairfuzz improves, which fits the claim that Fairfuzz prioritizes rare branches while fuzzing. TortoiseFuzz also ranks better,

Table 5. Observations on data-flow complexity features. PU(Previously Unknown), EP(Expected), UE(Unexpected).

FID	Fuzzer	Observation	State
D1, 3	AFL, AFLFast, EcoFuzz, Fairfuzz	Perform worse on programs with longer input strings.	EP
	AFL, AFLFast, EcoFuzz, MOpt	Perform worse on programs with deeper nested checks.	EP
	Fairfuzz	Perform better on programs with deeper nested checks.	EP
	Memlock (stack), TortoiseFuzz, Honggfuzz	Perform better on programs with longer input strings.	PU
	Memlock (stack), Honggfuzz	Perform excellently on programs with deeper nested checks.	PU
	Memlock (heap)	Perform worse on programs with deeper nested checks.	PU
	RedQueen, Laf-intel	Perform excellently/well where bug is guarded by long magic string.	EP
	Honggfuzz	Perform excellently where bug is guarded by long magic string.	PU
	Fuzzers other than RedQueen, Laf-intel, and Honggfuzz	Perform worse where bug is guarded by long magic string.	EP
D2, 3	RedQueen	Perform well with high counts and deep nesting of checksums.	EP
	Honggfuzz, Memlock	Perform excellently with high counts and deep nesting of checksums.	PU

while Memlock (stack) and Honggfuzz still maintain the best performance across all programs in this group. Interestingly, the other variant of Memlock, Memlock (heap), timed out at the nesting level of 5, suggesting that stack memory usage is more effective in guiding fuzzing process than heap memory usage in this feature group.

Figure 8(c) shows the median runtime of each fuzzer on programs with different lengths of magic strings. Most fuzzers were unable to detect bugs in programs with a magic string length of 3. Only three fuzzers, Honggfuzz, RedQueen, and Laf-intel, successfully found bugs in all benchmark programs. RedQueen and Honggfuzz performed particularly well on programs where the bug was guarded by a very long magic string (length of 10). Laf-intel was relatively slow compared to the other two. Memlock-heap had the worst performance, only making it past the first program. Its variant, Memlock (stack), showed slightly better performance but still timed out when the magic string length reached 3. This suggests that while using memory consumption to guide fuzzing process can be effective for finding bugs in programs with memory error-prone features, it struggles to handle complex constraints, such as long magic strings. AFL, AFLFast, EcoFuzz, FairFuzz and MOpt all stopped at a magic string length of 3, however, showing slightly better performance than AFL++ and TortoiseFuzz, which only managed to pass the magic string length of 2. We summarize the discussed observations across fuzzers in Table 5, and denote the features *magic bytes* and *nested magic bytes* as `D1` and `D3` in **FID** column.

Checksum test. Figures 8(d) and 8(e) illustrate the median runtimes of each fuzzer on programs with varying counts and depths of checksum tests, respectively. Most fuzzers experience a steady runtime increase as the count and depth of checksum tests grow. Notably, RedQueen manages high counts and deep nesting effectively, aligning with its design for handling complicated (nested) hard checks. However, Honggfuzz achieved the best performance overall, with Memlock (stack and heap) performing the second best, indicating their strong handling of complex checksum tests despite not being specifically tailored for such tasks. We summarize the discussed observations across fuzzers in Table 5, and denote features *checksum test* and *nested checksum test* as `D2` and `D3` in **FID** column.

6 Threats to Validity

Our work has several potential threats to validity. First, the papers we reviewed to extract program features are not exhaustive. We focused on grey-box fuzzing papers published within the last three years and on the most cited fuzzers from earlier years. Thus, other relevant papers may have been missed, potentially leading to missing important features. Additionally, the extracted features are based on the capabilities of current fuzzing techniques and do not account for future advancements that may introduce new features or surpass the capabilities of existing techniques. As the first step towards a feature-based fuzzing benchmark suite, we have developed and experimented with several features that have resulted in important insights. Second, the programs we generated in `FeatureBench` may not fully capture all aspects of program behavior related to control-flow and data-flow that can impact fuzzing performance. While we aimed to include a broad range of features, some might have been overlooked, which could limit the generality of our findings. Third, the generated programs are small and synthetically created, and do not comprehensively represent the real world faults. Therefore, these programs should be used in combination with other real world datasets for a useful evaluation.

7 Related Work

Fuzzing evaluation. Klees et al. [18] and Böhme et al. [5] analyzed the influence of various aspects of experimental setups and provided recommendations for more rigorous fuzzing evaluations. Building on this, Schloegel et al.

[44] examined the extent to which the guidelines proposed by Klees et al. [18] are followed in practice and introduced further refinements, such as emphasizing the importance of reproducible artifacts and well-established metrics in fuzzing evaluations. SENF [37] explored the impact of different evaluation parameters (e.g., the number of repetitions and runtime) and external factors (e.g., compiler settings and seed selection) on overall fuzzer performance. Fioraldi et al. [11] investigated the effect of specific internal fuzzing mechanisms, such as power schedules and search strategies, on fuzzer effectiveness. While these studies offer new insights into fuzzer performance, they still do not account for how specific program characteristics might influence fuzzer efficiency. Kummita et al. [19, 21] proposes to evaluate the fuzzers by visualizing the internals of fuzzing. Such visualization may complement our work and enhance the understanding of different fuzzers' performance in `FeatureBench`.

Program features and fuzzing. Wolff et al. [52] evaluated four fuzzers with respect to four program properties, namely, program size, proportion of equalities and inequalities, and proportion of shared library calls. They concluded that only the program size is relevant in influencing the performance of fuzzing. LEOPARD [9] uses program metrics to identify potential vulnerable functions in the program to support manual audits and fuzzing. The authors use complexity metrics (cyclomatic complexity, loop structure) and vulnerability metrics (number of parameters, pointers, control structures) to compute vulnerability scores of each function. These approaches do not concentrate on generating programs based on the configurable program features.

Zhu et al. [60] generated corpora for fuzzing evaluation based on search-hampering features (execution paths, magic values, checksums), which is the closest to our work. They extracted real-world program structures from GitHub, inserted bug contexts into these structures, and added extra code to ensure compilability. To the best of our knowledge, the corpora generated by Zhu et al. [60] are not publicly available, leaving the statistics of the generated programs unknown. Compared to their approach, our benchmark provides a more comprehensive set of features and implements each program feature with finer granularity. We designed quantifiable parameters to systematically control the strength of each feature from multiple aspects (e.g., controlling magic bytes by start index, length, and nesting level). This flexibility enables more precise program construction and allows for a detailed analysis of each feature's impact on fuzzing performance. Additionally, we evaluated 11 fuzzers on our benchmark, offering a more extensive comparison of performance across different fuzzers.

Program feature-based benchmarking. Several works have generated feature-based benchmarks in other software applications. Kummita et al. [20] created a microbenchmark of 49 test programs across 13 Python language features to evaluate Python call graph generation algorithms. Reif et al. [41] created a benchmark based on Java language features that contains 122 test cases across 23 language features to evaluate Java call graph analyses. TypeEvalPy[47] uses 154 programs to evaluate Python type inference tools. DroidBench [2] is an open-source microbenchmark consisting of 120 Android applications grouped into 13 categories, designed to evaluate taint analysis tools for Android applications on different static analysis algorithms. The RERS suite [42] is another similarly constructed benchmark suite designed for model-checking tools, aiming to develop a set of challenges in formal methods. It incorporates scalable complexity based on known properties during program generation process, producing small, medium and large programs for benchmarking [15]. Our work generates feature-based benchmarks for fuzzing which also incorporates scalable complexity based on extracted program features. Unlike RERS suite, we do not generate programs of varying sizes but instead provide fine-grained parameters to control the feature strength during benchmark generation. The varying parameter strengths allowed us to directly assess the impact of corresponding features on fuzzing performance, as illustrated in Section 5.3 (Figures 7 and 8).

8 Conclusions and Future Work

In this paper, we present a novel benchmark to evaluate fuzzers based on configurable program features. By reviewing 25 recent grey-box fuzzing papers, we extracted 7 program features associated with control-flow and data-flow that can impact fuzzer performance. Based on these features, we designed 10 configurable parameters that allow fine-grained control over program construction based on the strengths of various features. Our benchmark, `FeatureBench`, consists of 153 programs, and we evaluated 11 fuzzers using this benchmark. Our findings show that fuzzer performance varies significantly depending on program features and the strength of those features, highlighting the importance of considering program characteristics in fuzzing evaluations. Moving forward, we plan to extend our benchmark by adding new features such as algorithmic complexity and different bug types. In addition, we plan to perform static analysis to extract additional program features from real-world programs, and expand our benchmark to include features representing a broader range of real-world scenarios.

9 Data Availability

We have made `FeatureBench`, experimental data, and visualizations available in our artifact [1].

References

[1] Anonymous. 2024. Artifacts for the paper "Program Feature-based Benchmarking for Fuzz Testing". https://anonymous.4open.science/r/ISSTA2025-7C94/.

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM sigplan notices* 49, 6 (2014), 259–269.

[3] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence.. In *NDSS*, Vol. 19. 1–15.

[4] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1032–1043.

[5] Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the reliability of coverage-based fuzzer benchmarking. In *Proceedings of the 44th International Conference on Software Engineering*. 1621–1633.

[6] DARPA CGC. 2018. Darpa Cyber Grand Challenge (CGC). https://github.com/CyberGrandChallenge/.

[7] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 711–725.

[8] 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 *2016 IEEE Symposium on Security and Privacy (SP)*. 110–121. https://doi.org/10.1109/SP.2016.15

[9] Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. 2019. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 60–71.

[10] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. {AFL++}: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*.

[11] Andrea Fioraldi, Alessandro Mantovani, Dominik Maier, and Davide Balzarotti. 2023. Dissecting american fuzzy lop: a fuzzbench evaluation. *ACM transactions on software engineering and methodology* 32, 2 (2023), 1–26.

[12] FuzzBench. 2020. FuzzBench: 2020-09-07 report. https://www.fuzzbench.com/reports/sample/index.html.

[13] Google. 2014. Honggfuzz. https://github.com/google/honggfuzz.

[14] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 3, Article 49 (Nov. 2020), 29 pages. https://doi.org/10.1145/3428334

[15] Falk Howar, Marc Jasper, Malte Mues, David Schmidt, and Bernhard Steffen. 2021. The RERS challenge: towards controllable and scalable benchmark synthesis. *International Journal on Software Tools for Technology Transfer* 23, 6 (2021), 917–930.

[16] Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. 2020. Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1613–1627.

[17] Patrick Jauernig, Domagoj Jakobovic, Stjepan Picek, Emmanuel Stapf, and Ahmad-Reza Sadeghi. 2022. DARWIN: Survival of the fittest fuzzing mutators. *arXiv preprint arXiv:2210.11783* (2022).

[18] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2123–2138.

[19] Sriteja Kummita, Miao Miao, Eric Bodden, and Shiyi Wei. 2024. Visualization Task Taxonomy to Understand the Fuzzing Internals (Registered Report). In *Proceedings of the 3rd ACM International Fuzzing Workshop* (Vienna, Austria) *(FUZZING 2024)*. Association for Computing Machinery, New York, NY, USA, 13–22. https://doi.org/10.1145/3678722.3685530

[20] Sriteja Kummita, Goran Piskachev, Johannes Späth, and Eric Bodden. 2021. Qualitative and Quantitative Analysis of Callgraph Algorithms for Python. In *2021 International Conference on Code Quality (ICCQ)*. 1–15. https://doi.org/10.1109/ICCQ51190.2021.9392986

[21] Sriteja Kummita, Zenong Zhang, Eric Bodden, and Shiyi Wei. 2024. Visualizing and Understanding the Internals of Fuzzing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 2199–2204. https://doi.org/10.1145/3691620.3695284

[22] Laf-intel. 2016. Circumventing Fuzzing Roadblocks with Compiler Transformations. https://lafintel.wordpress.com/.

[23] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 75–86.

[24] Myungho Lee, Sooyoung Cha, and Hakjoo Oh. 2023. Learning seed-adaptive mutation strategies for greybox fuzzing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 384–396.

[25] Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*. 475–485.

[26] 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 2017 11th joint meeting on foundations of software engineering*. 627–637.

[27] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, et al. 2021. {UNIFUZZ}: A holistic and pragmatic {Metrics-Driven} platform for evaluating fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*. 2777–2794.

[28] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. 2022. Pata: Fuzzing with path aware taint analysis. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–17.

[29] Xuwei Liu, Wei You, Zhuo Zhang, and Xiangyu Zhang. 2022. TensileFuzz: facilitating seed input generation in fuzzing via string constraint solving. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 391–403.

[30] LLVM. 2015. LibFuzzer. https://llvm.org/docs/LibFuzzer.html.

[31] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*. 1949–1966.

[32] Chenyang Lyu, Hong Liang, Shouling Ji, Xuhong Zhang, Binbin Zhao, Meng Han, Yun Li, Zhe Wang, Wenhai Wang, and Raheem Beyah. 2022. SLIME: program-sensitive energy allocation for fuzzing. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*. 365–377.

[33] Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.

[34] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 1393–1403.

[35] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. FuzzBench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1393–1403. https://doi.org/10.1145/3468264.3473932

[36] Jiradet Ounjai, Valentin Wüstholz, and Maria Christakis. 2023. Green Fuzzer Benchmarking. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1396–1406.

[37] David Paaßen, Sebastian Surminski, Michael Rodler, and Lucas Davi. 2021. My fuzzer beats them all! developing a framework for fair evaluation and comparison of fuzzers. In *Computer Security–ESORICS 2021: 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4–8, 2021, Proceedings, Part I 26*. Springer, 173–193.

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

[39] Mohit Rajpal, William Blum, and Rishabh Singh. 2017. Not all bytes are equal: Neural byte sieve for fuzzing. *arXiv preprint arXiv:1711.04596* (2017).

[40] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware evolutionary fuzzing.. In *NDSS*, Vol. 17. 1–14.

[41] Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, and Mira Mezini. 2019. Judge: Identifying, understanding, and evaluating sources of unsoundness in call graphs. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 251–261.

[42] RERS. 2022. The RERS Challenge. https://rers-challenge.org/.

[43] Seemanta Saha, Laboni Sarker, Md Shafiuzzaman, Chaofan Shou, Albert Li, Ganesh Sankaran, and Tevfik Bultan. 2023. Rare path guided fuzzing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1295–1306.

[44] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. 2024. Sok: Prudent evaluation practices for fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1974–1993.

[45] SciPy. 2024. Statistics (scipy.stats). https://docs.scipy.org/doc/scipy/tutorial/stats.html.

[46] Kostya Serebryany. 2017. OSS-Fuzz - Google's continuous fuzzing service for open source software. USENIX Association, Vancouver, BC.

[47] Ashwin Prasad Shivarpatna Venkatesh, Samkutty Sabu, Jiawei Wang, Amir M. Mir, Li Li, and Eric Bodden. 2024. TypeEvalPy: A Micro-benchmarking Framework for Python Type Inference Tools. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings* (Lisbon, Portugal) *(ICSE-Companion '24)*. Association for Computing Machinery, New York, NY, USA, 49–53. https://doi.org/10.1145/3639478.3640033

[48] Charles Spearman. 1961. The proof and measurement of association between two things. (1961).

[49] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 497–512.

[50] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization.. In *NDSS*.

[51] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. Memlock: Memory usage guided fuzzing. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 765–777.

[52] Dylan Wolff, Marcel Böhme, and Abhik Roychoudhury. 2022. Explainable fuzzer evaluation. *arXiv preprint arXiv:2212.09519* (2022).

[53] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. 2020. {EcoFuzz}: Adaptive {Energy-Saving} greybox fuzzing as a variant of the adversarial {Multi-Armed} bandit. In *29th USENIX Security Symposium (USENIX Security 20)*. 2307–2324.

[54] Michał Zalewski. 2013. American Fuzzy Lop (2.52b). https://lcamtuf.coredump.cx/afl/.

[55] Gen Zhang, Pengfei Wang, Tai Yue, Xiangdong Kong, Shan Huang, Xu Zhou, and Kai Lu. 2024. Mobfuzz: Adaptive multi-objective optimization in gray-box fuzzing. *arXiv preprint arXiv:2401.15956* (2024).

[56] Kunpeng Zhang, Xiaogang Zhu, Xi Xiao, Minhui Xue, Chao Zhang, and Sheng Wen. 2023. SHAPFUZZ: Efficient Fuzzing via Shapley-Guided Byte Selection. *arXiv preprint arXiv:2308.09239* (2023).

[57] Zenong Zhang, Zach Patterson, Michael Hicks, and Shiyi Wei. 2022. FIXREVERTER: A Realistic Bug Injection Methodology for Benchmarking Fuzz Testing. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 3699–3715. https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-zenong

[58] Lei Zhao, Yue Duan, and Jifeng XUAN. 2019. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. Network and Distributed System Security Symposium (NDSS).

[59] Xiaoqi Zhao, Haipeng Qu, Wenjie Lv, Shuo Li, and Jianliang Xu. 2021. MooFuzz: many-objective optimization seed schedule for fuzzer. *Mathematics* 9, 3 (2021), 205.

[60] Xiaogang Zhu, Xiaotao Feng, Tengyun Jiao, Sheng Wen, Yang Xiang, Seyit Camtepe, and Jingling Xue. 2019. A feature-oriented corpus for understanding, evaluating and improving fuzz testing. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. 658–663.